12	0	(((((ODBC and (DBMS or "database management system")) and (pars$3 with code) and (sql with statement)) and (sql with ODBC)) not ((((ODBC and (DBMS or "database management system")) and (pars$3 with code) and (sql with statement)) and (sql with ODBC)) and (ODBC with call))) and "source code") and (convert$3 with sql) same ODBC	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 17:46
13	30	(((((ODBC and (DBMS or "database management system")) and (pars$3 with code) and (sql with statement)) and (sql with ODBC)) not ((((ODBC and (DBMS or "database management system")) and (pars$3 with code) and (sql with statement)) and (sql with ODBC)) and (ODBC with call))) and "source code") and (convert$3 with sql)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 17:47
14	0	(((((ODBC and (DBMS or "database management system")) and (pars$3 with code) and (sql with statement)) and (sql with ODBC)) not ((((ODBC and (DBMS or "database management system")) and (pars$3 with code) and (sql with statement)) and (sql with ODBC)) and (ODBC with call))) and "source code") and (convert$3 with ODBC)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 17:47
15	30	((((((ODBC and (DBMS or "database management system")) and (pars$3 with code) and (sql with statement)) and (sql with ODBC)) not ((((ODBC and (DBMS or "database management system")) and (pars$3 with code) and (sql with statement)) and (sql with ODBC)) and (ODBC with call))) and "source code") and (convert$3 with sql)) and call and function	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 17:51
16	12	(sql and odbc).ab.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 17:52
-	1139	SQL and ODBC	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 15:15
-	142	(SQL and ODBC) and (pars$ with code)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 15:16
-	68	((SQL and ODBC) and (pars$ with code)) and keyword$1	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 15:16
-	43	(((SQL and ODBC) and (pars$ with code)) and keyword$1) and (sql with statement)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 15:16
-	3	((((SQL and ODBC) and (pars$ with code)) and keyword$1) and (sql with statement)) and (ODBC with call) and (function with call)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 15:44

-	4	((((SQL and ODBC) and (pars$ with code)) and keyword$1) and (sql with statement)) and (ODBC with call)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 16:41
-	36	(ODBC and (DBMS or "database management system")) and (ODBC with function)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 17:40
-	4	((ODBC and (DBMS or "database management system")) and (ODBC with function)) and (convert$3 with (SQL or "structure queried language") with ODBC)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/02/07 16:44

US 20030046673A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2003/0046673 A1

Copeland et al. (43) Pub. Date: Mar. 6, 2003

(54) LINKTIME RECOGNITION OF ALTERNATIVE IMPLEMENTATIONS OF PROGRAMMED FUNCTIONALITY

(75) Inventors: Jeffrey L. Copeland, Bellevue, WA (US); Jason D. Zions, Bellevue, WA (US); Donn S. Terry, Woodinville, WA (US)

Correspondence Address:
LEYDIG VOIT & MAYER, LTD
TWO PRUDENTIAL PLAZA, SUITE 4900
180 NORTH STETSON AVENUE
CHICAGO, IL 60601-6780 (US)

(73) Assignee: Microsoft Corporation, One Microsoft Way, Redmond, WA 98052 (US)

(21) Appl. No.: 09/897,540

(22) Filed: Jun. 29, 2001

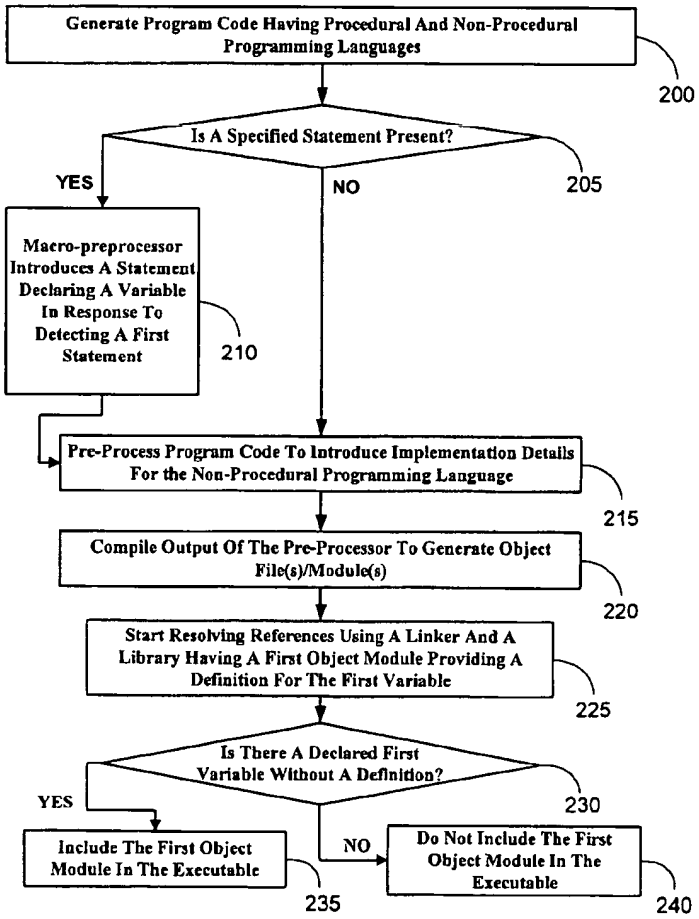
Publication Classification

(51) Int. Cl.[7] G06F 9/44

(52) U.S. Cl. 717/163

(57) ABSTRACT

A method and apparatus are disclosed for optimizing the runtime behavior of database or other applications by allowing selection of alternative code segments during linking of pre-compiled object modules. A macro-preprocessor inserts a declaration for a global variable in the source code in response to an occurrence of a command of interest. The linker selects object modules for executing other commands based on the presence or absence of the declaration for the global variable in the preprocessed source code. The method and apparatus are useful in implementing programming language statements including non-procedural programming languages such as the Embedded Structured Query Language (ESQL).



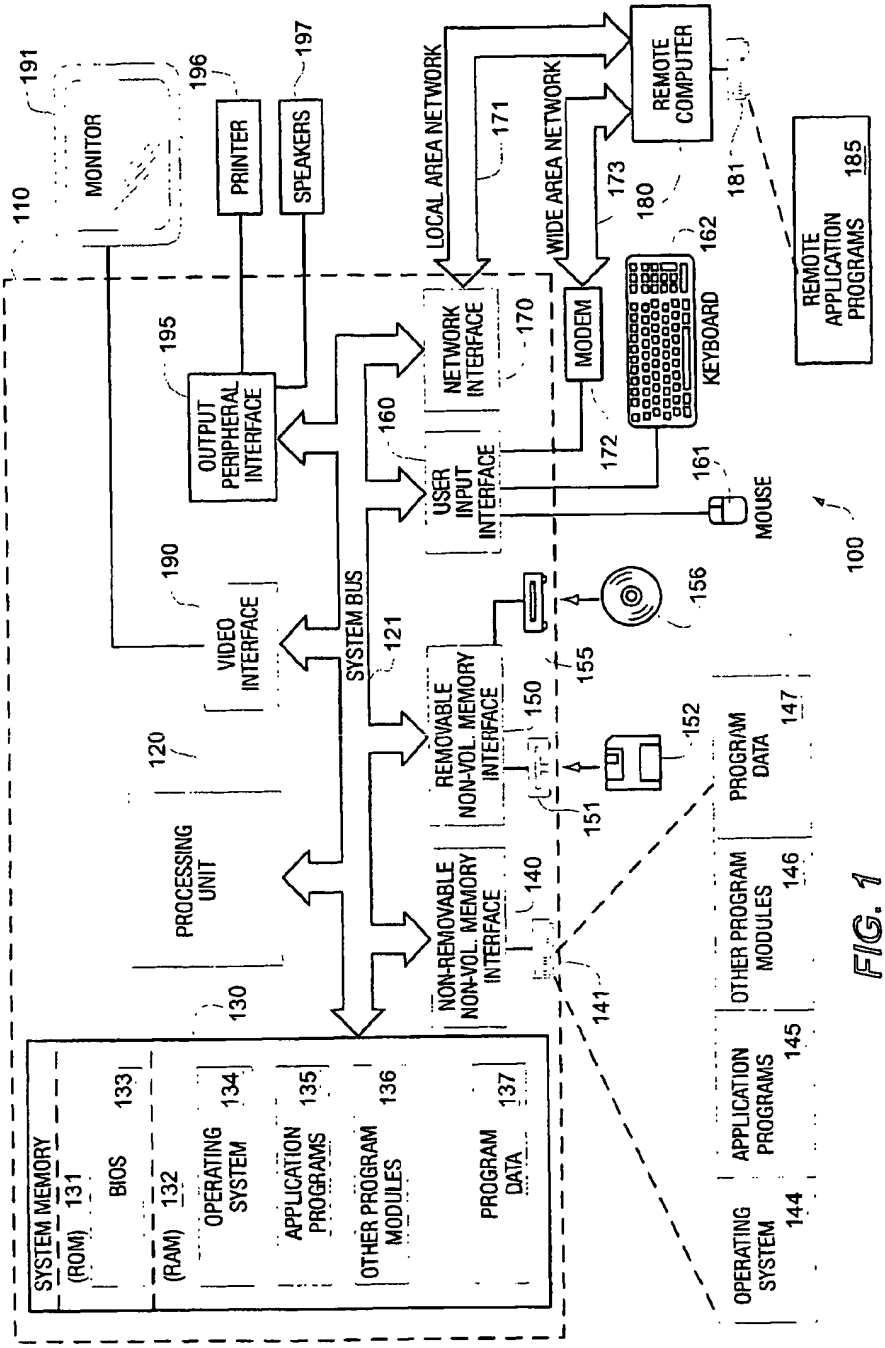

SYSTEM MEMORY
(ROM) 131
BIOS 133
(RAM) 132
OPERATING SYSTEM 134
APPLICATION PROGRAMS 135
OTHER PROGRAM MODULES 136
PROGRAM DATA 137
PROCESSING UNIT 120
VIDEO INTERFACE 190
OUTPUT PERIPHERAL INTERFACE 195
NON-REMOVABLE NON-VOL. MEMORY INTERFACE 140
REMOVABLE NON-VOL. MEMORY INTERFACE 150
USER INPUT INTERFACE 160
NETWORK INTERFACE 170
SYSTEM BUS 121
MONITOR 191
PRINTER 196
SPEAKERS 197
LOCAL AREA NETWORK 171
WIDE AREA NETWORK 173
MODEM 172
KEYBOARD 162
MOUSE 161
REMOTE COMPUTER 180
REMOTE APPLICATION PROGRAMS 185
OPERATING SYSTEM 144
APPLICATION PROGRAMS 145
OTHER PROGRAM MODULES 146
PROGRAM DATA 147
110
100
130
141
151
152
155
156
181

FIG. 1

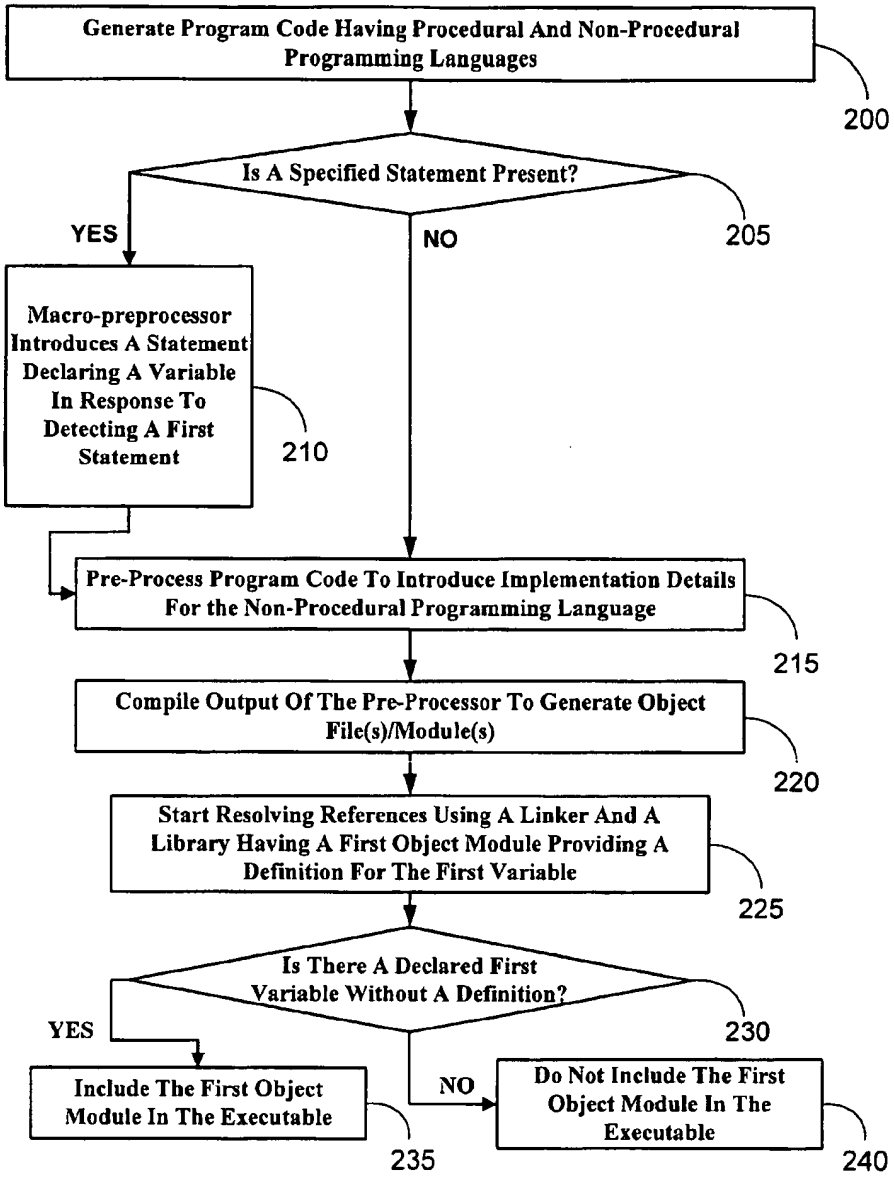

Generate Program Code Having Procedural And Non-Procedural Programming Languages
200
Is A Specified Statement Present?
205
YES
NO
Macro-preprocessor Introduces A Statement Declaring A Variable In Response To Detecting A First Statement
210
Pre-Process Program Code To Introduce Implementation Details For the Non-Procedural Programming Language
215
Compile Output Of The Pre-Processor To Generate Object File(s)/Module(s)
220
Start Resolving References Using A Linker And A Library Having A First Object Module Providing A Definition For The First Variable
225
Is There A Declared First Variable Without A Definition?
230
YES
NO
Include The First Object Module In The Executable
235
Do Not Include The First Object Module In The Executable
240

FIGURE 2

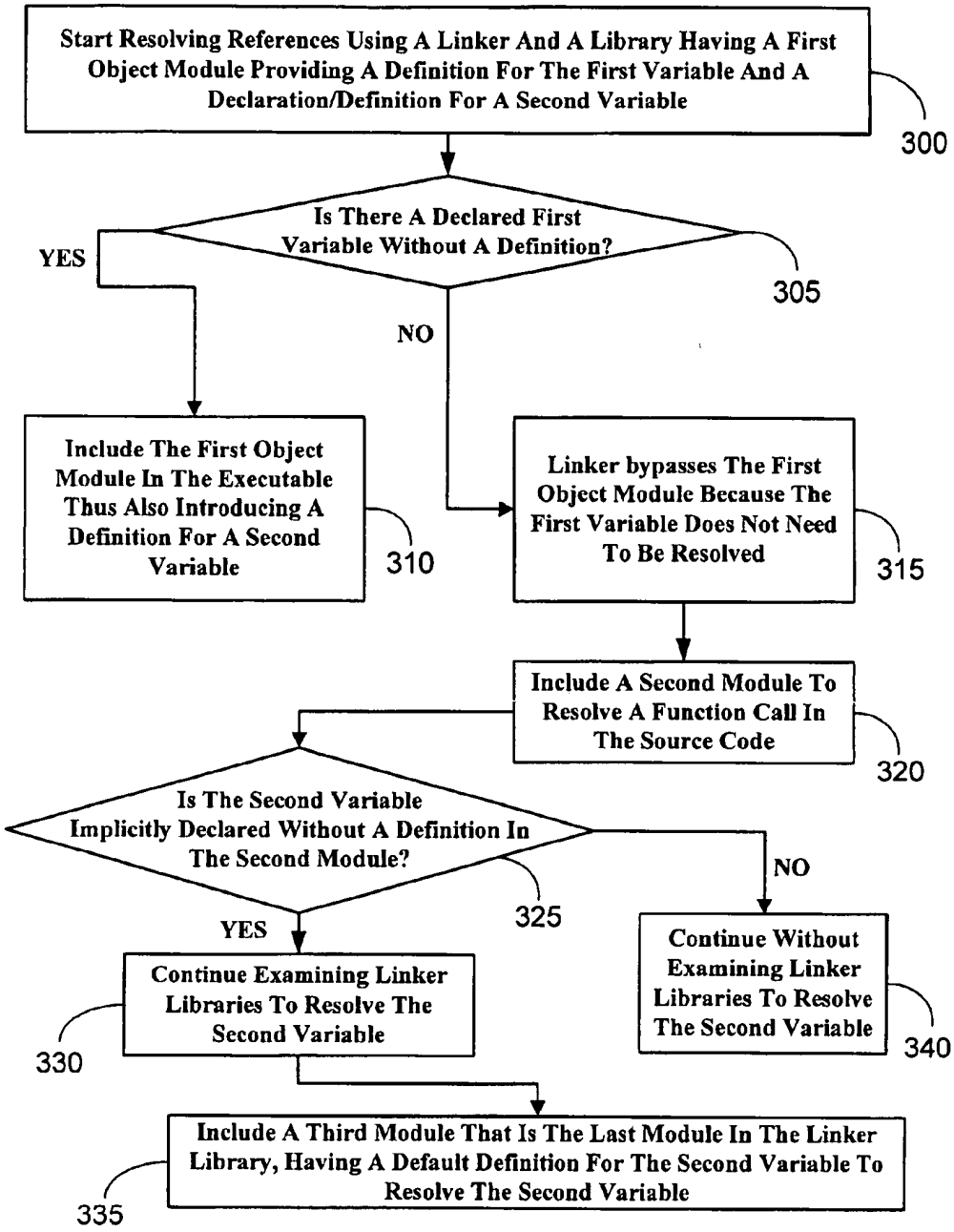

Start Resolving References Using A Linker And A Library Having A First Object Module Providing A Definition For The First Variable And A Declaration/Definition For A Second Variable
300
Is There A Declared First Variable Without A Definition?
305
YES
NO
Include The First Object Module In The Executable Thus Also Introducing A Definition For A Second Variable
310
Linker bypasses The First Object Module Because The First Variable Does Not Need To Be Resolved
315
Include A Second Module To Resolve A Function Call In The Source Code
320
Is The Second Variable Implicitly Declared Without A Definition In The Second Module?
325
YES
NO
Continue Examining Linker Libraries To Resolve The Second Variable
330
Continue Without Examining Linker Libraries To Resolve The Second Variable
340
Include A Third Module That Is The Last Module In The Linker Library, Having A Default Definition For The Second Variable To Resolve The Second Variable
335

FIGURE 3

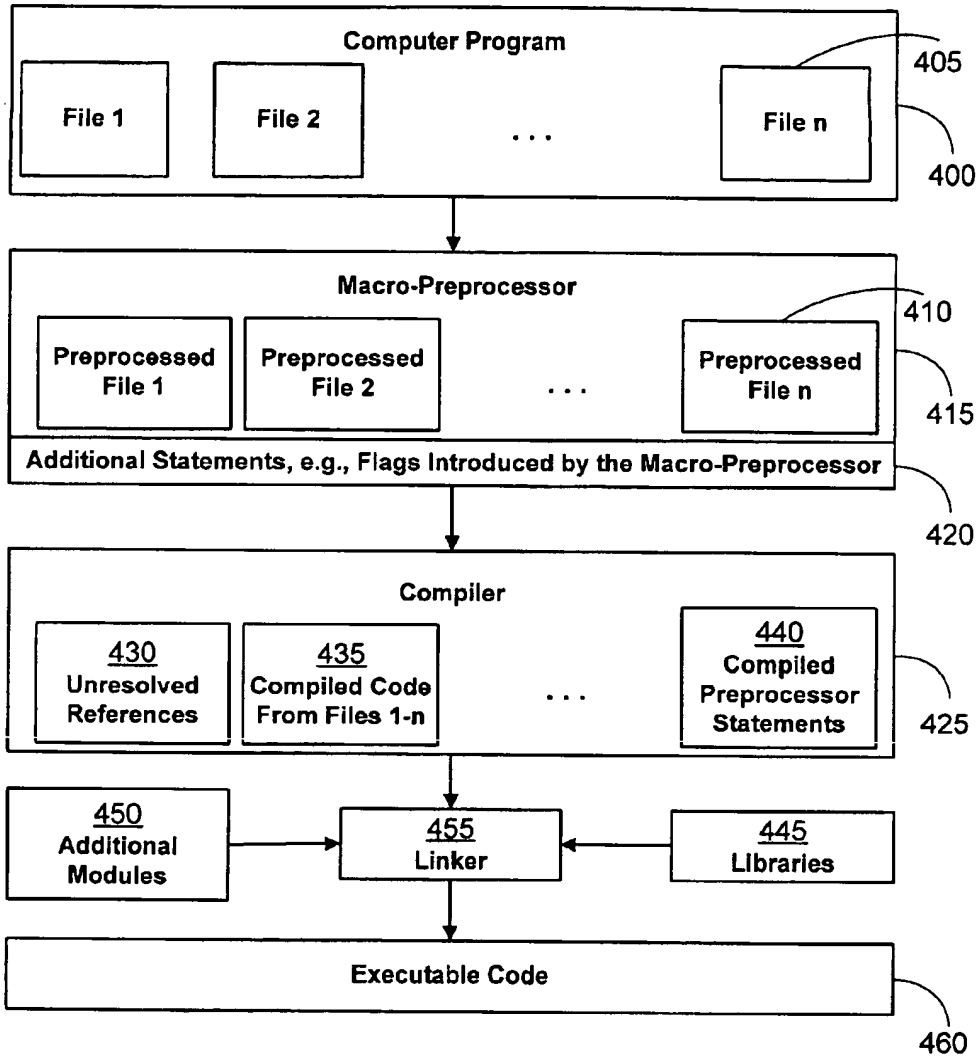

Computer Program
File 1
File 2
...
File n
405
400
Macro-Preprocessor
Preprocessed File 1
Preprocessed File 2
...
Preprocessed File n
410
415
Additional Statements, e.g., Flags Introduced by the Macro-Preprocessor
420
Compiler
430 Unresolved References
435 Compiled Code From Files 1-n
...
440 Compiled Preprocessor Statements
425
450 Additional Modules
455 Linker
445 Libraries
Executable Code
460

FIGURE 4

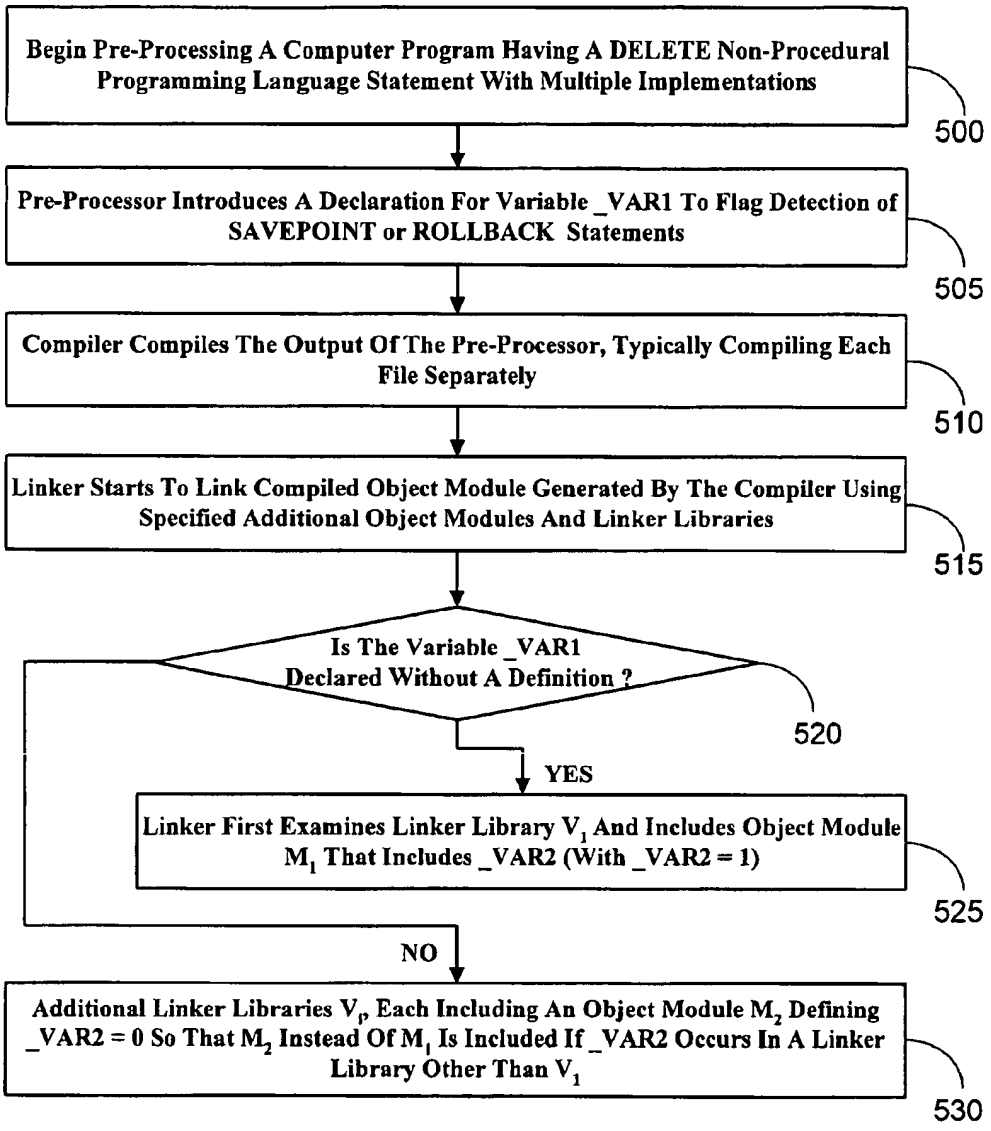

Begin Pre-Processing A Computer Program Having A DELETE Non-Procedural Programming Language Statement With Multiple Implementations
500
Pre-Processor Introduces A Declaration For Variable _VAR1 To Flag Detection of SAVEPOINT or ROLLBACK Statements
505
Compiler Compiles The Output Of The Pre-Processor, Typically Compiling Each File Separately
510
Linker Starts To Link Compiled Object Module Generated By The Compiler Using Specified Additional Object Modules And Linker Libraries
515
Is The Variable _VAR1 Declared Without A Definition ?
520
YES
Linker First Examines Linker Library V_1 And Includes Object Module M_1 That Includes _VAR2 (With _VAR2 = 1)
525
NO
Additional Linker Libraries V_p Each Including An Object Module M_2 Defining _VAR2 = 0 So That M_2 Instead Of M_1 Is Included If _VAR2 Occurs In A Linker Library Other Than V_1
530

FIGURE 5

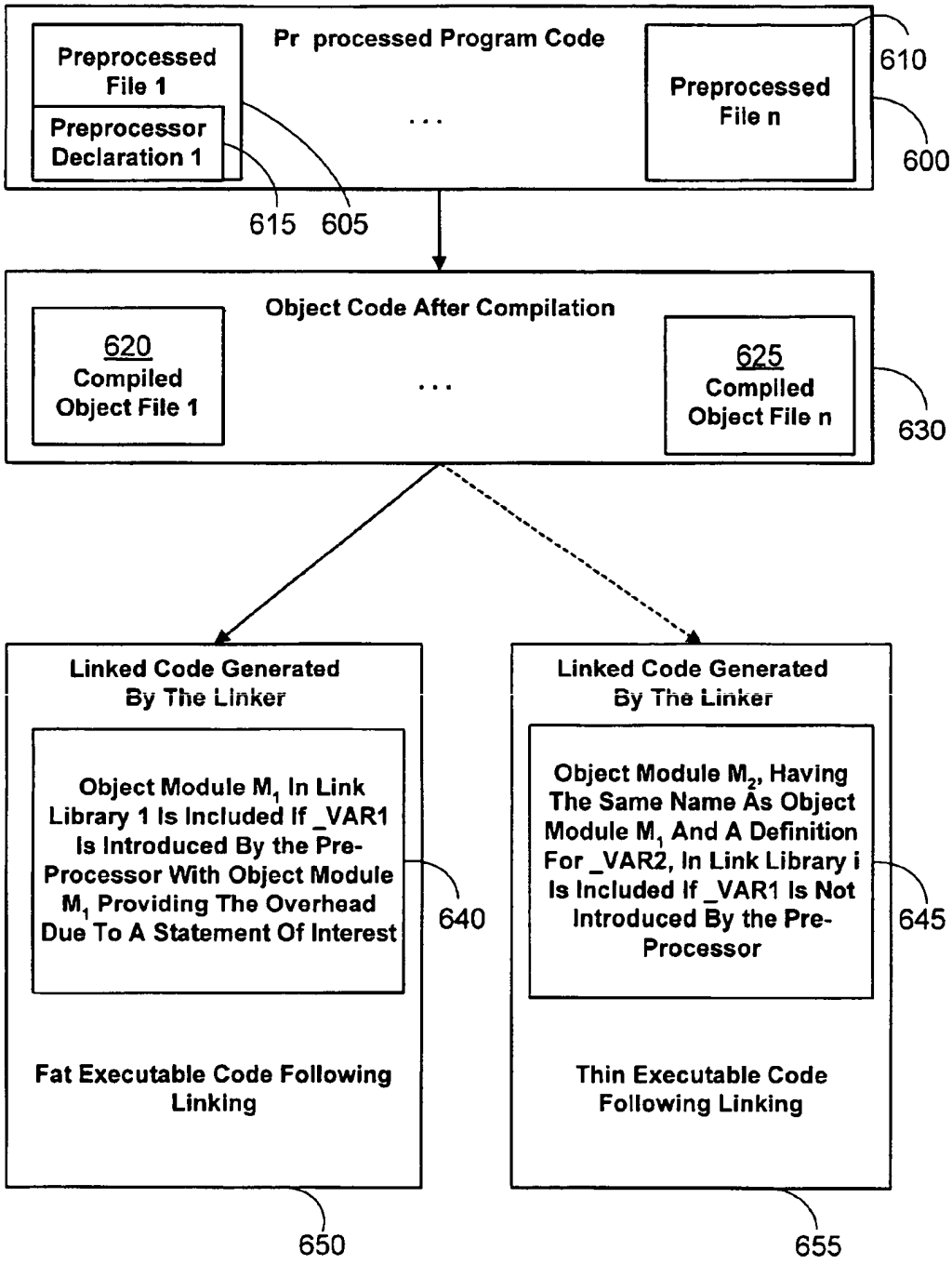

Pr processed Program Code
Preprocessed File 1
Preprocessor Declaration 1
Preprocessed File n
610
600
615
605
Object Code After Compilation
620
Compiled Object File 1
625
Compiled Object File n
630
Linked Code Generated By The Linker
Object Module M_1 In Link Library 1 Is Included If _VAR1 Is Introduced By the Pre-Processor With Object Module M_1 Providing The Overhead Due To A Statement Of Interest
640
Fat Executable Code Following Linking
650
Linked Code Generated By The Linker
Object Module M_2, Having The Same Name As Object Module M_1 And A Definition For _VAR2, In Link Library i Is Included If _VAR1 Is Not Introduced By the Pre-Processor
645
Thin Executable Code Following Linking
655

FIGURE 6

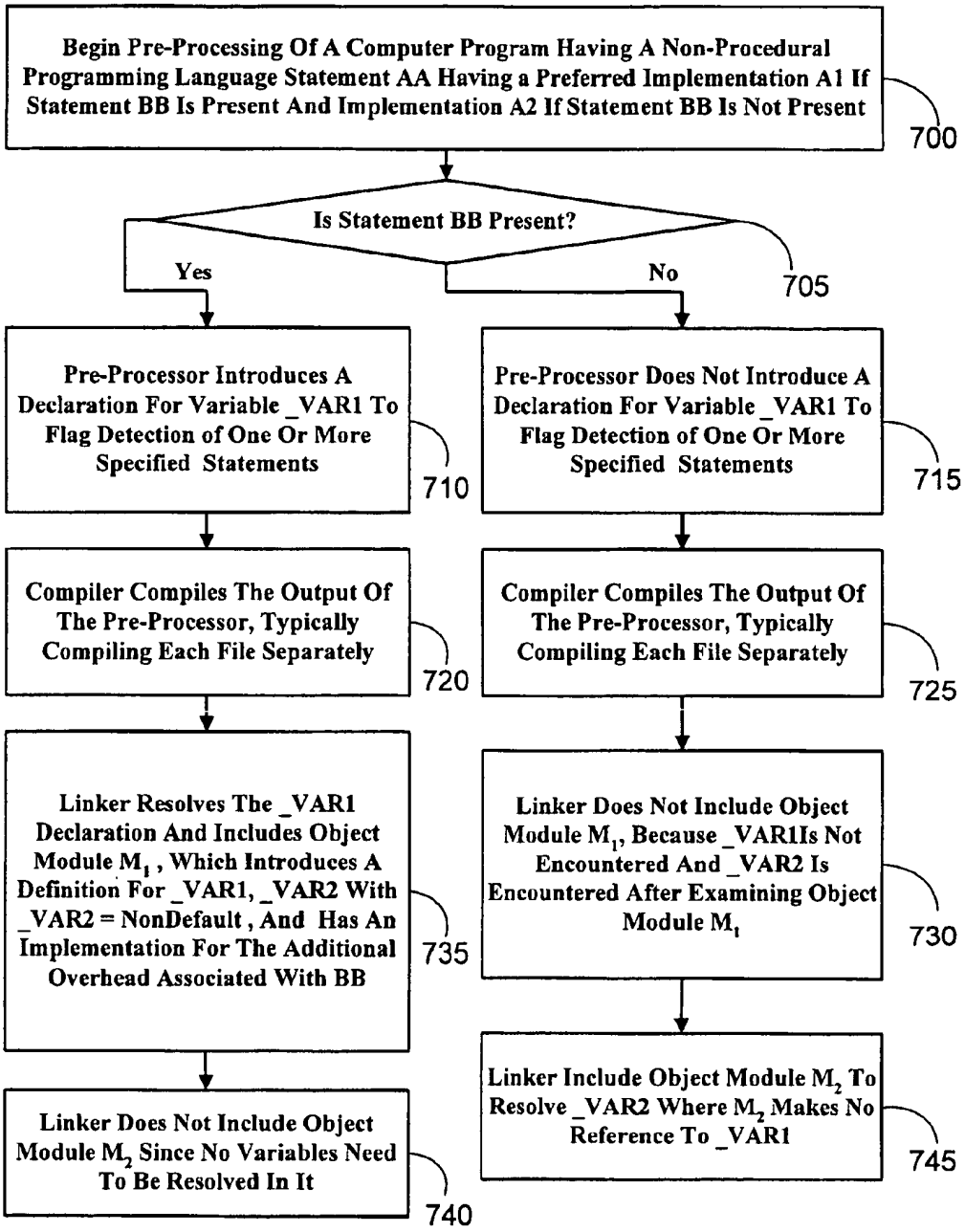

Begin Pre-Processing Of A Computer Program Having A Non-Procedural Programming Language Statement AA Having a Preferred Implementation A1 If Statement BB Is Present And Implementation A2 If Statement BB Is Not Present
700
Is Statement BB Present?
705
Yes
No
Pre-Processor Introduces A Declaration For Variable _VAR1 To Flag Detection of One Or More Specified Statements
710
Pre-Processor Does Not Introduce A Declaration For Variable _VAR1 To Flag Detection of One Or More Specified Statements
715
Compiler Compiles The Output Of The Pre-Processor, Typically Compiling Each File Separately
720
Compiler Compiles The Output Of The Pre-Processor, Typically Compiling Each File Separately
725
Linker Resolves The _VAR1 Declaration And Includes Object Module M_1, Which Introduces A Definition For _VAR1, _VAR2 With _VAR2 = NonDefault, And Has An Implementation For The Additional Overhead Associated With BB
735
Linker Does Not Include Object Module M_1, Because _VAR1Is Not Encountered And _VAR2 Is Encountered After Examining Object Module M_1
730
Linker Include Object Module M_2 To Resolve _VAR2 Where M_2 Makes No Reference To _VAR1
745
Linker Does Not Include Object Module M_2 Since No Variables Need To Be Resolved In It
740

FIGURE 7

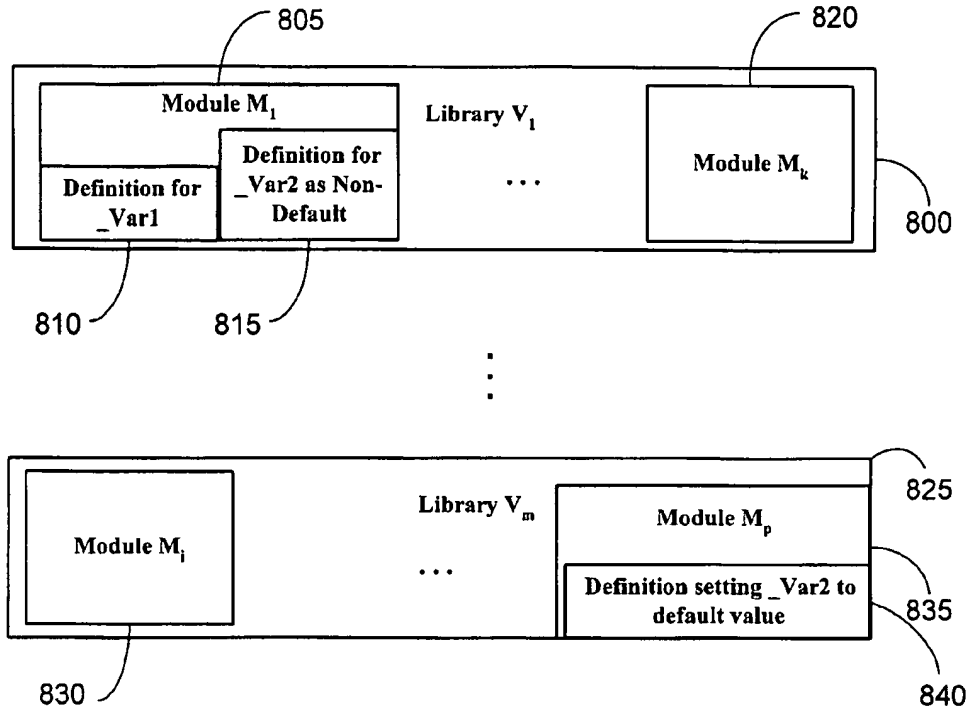

805
820
Module M_1
Library V_1
Definition for _Var2 as Non-Default
Definition for _Var1
Module M_k
800
810
815
Module M_i
Library V_m
Module M_p
Definition setting _Var2 to default value
825
835
830
840

FIGURE 8

LINKTIME RECOGNITION OF ALTERNATIVE IMPLEMENTATIONS OF PROGRAMMED FUNCTIONALITY

TECHNICAL FIELD OF THE INVENTION

[0001] This invention relates generally to generating an efficient executable corresponding to a program written in a higher-level computer programming language, and more particularly to use indicators that change the executable code to take into account the context in implementing a particular command.

BACKGROUND OF THE INVENTION

[0002] The advent of databases and e-commerce requires the ability to request services from a variety of databases without knowing the exact implementation of the database or of the statements used to request the services. These request statements are made in a non-procedural programming language that does not provide an explicit implementation. Instead, the developers of particular databases or non-procedural programming language statements provide proprietary implementations for the statements rendered in the non-procedural language.

[0003] Structured Query Language ("SQL") is an illustrative example of a non-procedural language. SQL differs from a procedural language like FORTRAN in that it does not specify how a particular request is carried out, but instead allows the database manager to provide the relevant details. Thus, a command in SQL merely states a request and not how it is carried out.

[0004] SQL includes: a Data Development Language ("DDL") for creating databases and data structures, but not necessarily data itself; a Data Manipulation Language ("DML") facilitating database maintenance and actual operations on data; and a Data Control Language ("DCL") for specifying security requirements. Some examples of SQL commands include the DDL commands CREATE, ALTER and DROP, DML statements and functions such as INSERT, UPDATE, DELETE, SELECT, COUNT, SUM and the like, and DCL commands such as COMMIT, ROLLBACK, GRANT and REVOKE.

[0005] SQL permits interactions with a database in an atomic manner, i.e. only one user may access a unit of data, to prevent other users from changing the database between operations constituting a transaction. The code used to implement these commands and functions is the responsibility of the database developer or vendor. Of course, universal support for SQL commands ensures that any user can access and use a SQL compliant database regardless of the database vendor and particular implementation details.

[0006] SQL commands such as COMMIT and ROLLBACK are of interest in an exemplary embodiment of the invention. These SQL commands protect a database against inadvertent corruption. To this end the database itself is not affected until the COMMIT command is given. If an error occurs then a ROLLBACK command restores the state of the system to that at the conclusion of the previous COMMIT command. A transaction is terminated by either a COMMIT command or ROLLBACK command combined with allowing other users access to the data. A ROLLBACK command requires buffering of all operations following a COMMIT command to permit restoration of the state following the COMMIT command.

[0007] If the transaction fails or a user cancels a transaction, a ROLLBACK results in clearing the buffered operations and removing access restrictions to restore the database to its state prior to the initiation of the now failed transaction. On the other hand, a COMMIT command results in updating the database followed by clearing of the buffered operations.

[0008] Another SQL command, SAVEPOINT, enables restoring the system to an earlier defined state that need not be the state at the conclusion of the previous COMMIT command. Like the COMMIT command in the context of the ROLLBACK command, SAVEPOINT provides a prior state of the system for the ROLLBACK command. Unlike the COMMIT command, however, the SAVEPOINT command does not require changes to the database. Instead SAVEPOINT enables specification of a defined state for system restoration. In some embodiments the SAVEPOINT command specifies multiple earlier states distinguished by their respective identifiers. If desired, the system can be restored to one of the specified earlier states by executing a ROLLBACK to the specified state. If a COMMIT command is given then all buffered operations are cleared along with the states specified by the SAVEPOINT command.

[0009] Implementing the SAVEPOINT or ROLLBACK commands requires considerable overhead since other commands must therefore provide buffering. On the other hand, it is not necessary to support buffering if the SAVEPOINT or ROLLBACK commands are not used. A typical application includes SQL statements in several files and a compiler compiles only one file at a time. Thus, it is not possible to decide when compiling a particular file whether buffering-related code is needed due to a statement in another file.

[0010] SQL applications written using SQL statements and functions can be combined with source code in a programming language such as C++ in Embedded SQL ("ESQL"). An ESQL application can include several source code files. The source files for an ESQL application are preprocessed by a macro-preprocessor. Typically, the macro-preprocessor generates code for the various embedded SQL statements or introduces additional statements followed by a compiler compiling the output of the macro-preprocessor. Compiling a source file generates an object module corresponding to the source file. The linker links object modules to generate the executable program.

[0011] Compiling a source code file includes several operations. A compiler parses the source code, carries out several checks to ensure conformity with the programming language specifications and then translates the parsed code to generate a lower level code such as machine code for execution on a computer. In some instances, the code is assembly or byte code that needs further translation for actual execution on a particular computer. A compiler allocates memory for each variable to properly translate source code to generate executable code. The compiler allocates memory for each variable in accordance with a "type" specification for the variable in question.

[0012] Type information is specified in a "declaration" statement. Each variable is assigned a particular type. The compiler enters the type information for each variable into a symbol table associated with an object module. When

several object modules in the same executable share a variable it is important to ensure that only one module actually allocates memory for the variable. The compiler allocates memory in response to a "definition" statement for a particular variable. However, the declaration and/or definition statements are allowed to be implicit in many programming languages.

[0013] The "C" programming language permits an "extern" declaration in a source file that tells the compiler that memory for the specified variable is allocated in another file. Consequently, a C compiler only creates a variable entry in the symbol table that serves as a place holder for the variable but leaves the actual memory allocation to another file. The variable merely points to its entry in the symbol table and is redirected to the actual memory allocation following identification of the intended memory location. Thus, there are several declarations for a variable but there can be only one definition. No value can be assigned to a variable unless the variable is defined because there is no memory allocated to store it.

[0014] Following compilation, a linker links the resultant object files to generate the executable for the application. The linking may be static or dynamic. In static linking the object files identified by the linker for the resolution of all variables are copied to generate an executable file. In contrast, dynamic linking allows fetching an object file at either load time or at runtime. Consequently, the same object module is used by several applications. As is evident, typically dynamic linking results in lower memory requirements and smaller executable sizes. Furthermore, a programmer can modify and recompile a dynamically-linked module independent of another module, thus making software maintenance easier and less expensive.

[0015] Declaring a variable with an "extern" keyword requires the linker to identify the actual memory allocated for the variable in other object modules. To this end the linker searches symbol tables associated with object modules or libraries for a module providing a definition for the variable in question. This process is termed resolving the variable. Proper resolution of a variable is required before it can actually be used in an executable file.

[0016] In software development projects a software application is refined over the life of the project. Through the development process, concepts concerning various problems and solutions are often revised, and the functions and features of the final software application are often quite different from those at the beginning of the project. Support for additional features supporting execution of other statements in a non-procedural language statement reduces the execution efficiency of programs that do not use these additional features. On the other hand, adding distinct commands to provide the additional features results in complex programming languages with many statements differing only in the context in which they should be used. For example, if there is at least one command that requires buffering prior changes to a database in an SQL-based application, then implementations of other commands affecting the database need to support buffering. On the other hand, if no command requiring buffering is used in an application then the program overhead for buffering unnecessarily slows down the application.

[0017] As a programming language evolves to develop specific commands for a particular context, developers have to learn different commands for accomplishing similar tasks rather than preserving their existing familiarity with the programming tool. Similar sounding commands that differ in subtle but significant details increase the risk that a programmer inadvertently uses the less effective command. Such errors are difficult to identify since some may only sporadically result in bugs. Therefore, it is desirable to have a system and method for providing contextually efficient implementations for a programming language command that can be invoked automatically without requiring the programmer to use different commands to invoke optimized implementations for different contexts.

SUMMARY OF THE INVENTION

[0018] In view of the foregoing, the present invention provides a method and system for selecting one of several implementations of a higher level programming language statement in response to the occurrence or non-occurrence of another statement in a computer program. The invention enables transparent selection of contextually efficient code. Thus, users and developers need not use different higher language statements to invoke a context-specific implementation.

[0019] A macro-preprocessor enables choosing a context sensitive implementation for a higher language statement. The macro-preprocessor introduces a first global variable declaration in response to identifying a specific context. In an embodiment, the specific context is defined by the presence of one or more specified statements in a source file processed by the macro-preprocessor. In an embodiment of the invention, the first global variable enables setting a desired value for a second variable by introducing the second variable in a first object module supplying the definition for the first global variable. This strategy provides for a level of indirection to include the first object module in response to identifying a specified context. In the event the linker does not include the first object module, an alternative definition for the second variable is provided in a second object module.

[0020] In an embodiment of the invention a linker library object module is loaded using a wrapper based upon the first global variable. Moreover, an embodiment of the invention enables conditionally executing a first program sequence in response to the second variable value specifying a context of interest.

[0021] For instance, the need to support the underlying implementation of a RESTORE command may require a DELETE command to include buffering deleted data. However, if no RESTORE command is used in a program then there is no need to incur the overhead of buffering extensive information in the implementation of the DELETE command.

[0022] It should be noted that the invention, while illustrated with SQL, is not limited to SQL or even non-procedural languages, but instead includes higher-level languages and scripts. Such higher-level languages and scripts can benefit from using different binary, byte-code or macro implementations for the same command depending on a particular context.

[0023] Additional features and advantages of the invention will be made apparent from the following detailed

description of illustrative embodiments, which proceeds with reference to the accompanying figures.

BRIEF DESCRIPTION OF THE DRAWINGS

[0024] While the appended claims set forth the features of the present invention with particularity, the invention, together with its objects and advantages, may be best understood from the following detailed description taken in conjunction with the accompanying drawings of which:

[0025] **FIG. 1** is a block diagram generally illustrating an exemplary computing environment in which databases and other software structures are implemented along with higher-level languages being used to describe desired services, including services pertaining to the database;

[0026] **FIG. 2** is a flowchart summarizing an exemplary set of steps of preprocessing, compiling, linking and executing a computer program in a computing environment;

[0027] **FIG. 3** is a flowchart summarizing exemplary steps of an embodiment that introduces statements in the program code to generate an efficient runtime executable in accordance with the invention;

[0028] **FIG. 4** is a flow diagram illustratively depicting compilation of a program using a preprocessor, a compiler and a linker;

[0029] **FIG. 5** is a flow chart illustrating an exemplary set of steps for introducing a global variable reflecting the context of command in an embodiment of the invention;

[0030] **FIG. 6** is a flow diagram illustratively depicting transformation of a program from high level instructions to executable code, including transformation to the pre-processed code and subsequent incorporation of particular object modules based upon a detected context;

[0031] **FIG. 7** is a flow diagram illustrating the different implementations for a second statement due to the occurrence or non-occurrence of a first statement in a computer program in accordance with an embodiment of the invention; and

[0032] **FIG. 8** illustrates exemplary linker libraries.

DETAILED DESCRIPTION OF THE INVENTION

[0033] Turning to the drawings, wherein like reference numerals refer to like elements, the invention is illustrated as being implemented in a suitable computing environment. Although not required, the invention will be described in the general context of computer-executable instructions, such as program modules, being executed in a computing environment. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multi-processor systems, microprocessor based or programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

[0034] **FIG. 1** illustrates an example of a suitable computing system environment **100** on which the invention may be implemented. The computing system environment **100** is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the invention. Neither should the computing environment **100** be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary operating environment **100**.

[0035] The invention is operational with numerous other general-purpose or special-purpose computing system environments or configurations. Examples of well-known computing systems, environments, and configurations that may be suitable for use with the invention include, but are not limited to, personal computers, server computers, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable consumer electronics, network PCs, minicomputers, mainframe computers, and distributed computing environments that include any of the above systems or devices.

[0036] The invention may be described in the general context of computer-executable instructions, such as program modules, being executed by a computer. Generally, program modules include routines, programs, objects, components, data structures, etc., that perform particular tasks or implement particular abstract data types. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote computer storage media including memory storage devices.

[0037] With reference to **FIG. 1**, an exemplary system for implementing the invention includes a general-purpose computing device in the form of a computer **110**. Components of the computer **110** may include, but are not limited to, a processing unit **120**, a system memory **130**, and a system bus **121** that couples various system components including the system memory to the processing unit **120**. The system bus **121** may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus, also known as Mezzanine bus.

[0038] The computer **110** typically includes a variety of computer-readable media. Computer-readable media can be any available media that can be accessed by the computer **110** and include both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer-readable media may include computer storage media and communications media. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer-readable instructions, data structures, program modules, or other data.

Computer storage media include, but are not limited to, random-access memory (RAM), read-only memory (ROM), EEPROM, flash memory, or other memory technology, CD-ROM, digital versatile disks (DVD), or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage, or other magnetic storage devices, or any other medium which can be used to store the desired information and which can accessed by the computer **110**. Communications media typically embody computer-readable instructions, data structures, program modules, or other data in a modulated data signal such as a carrier wave or other transport mechanism and include any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communications media include wired media such as a wired network and a direct-wired connection and wireless media such as acoustic, RF, and infrared media. Combinations of the any of the above should also be included within the scope of computer-readable media.

[0039] The system memory **130** includes computer storage media in the form of volatile and nonvolatile memory such as ROM **131** and RAM **132**. A basic input/output system (BIOS) **133**, containing the basic routines that help to transfer information between elements within the computer **110**, such as during start-up, is typically stored in ROM **131**. RAM **132** typically contains data and program modules that are immediately accessible to or presently being operated on by processing unit **120**. By way of example, and not limitation, **FIG. 1** illustrates an operating system **134**, application programs **135**, other program modules **136**, and program data **137**. Often, the operating system **134** offers services to applications programs **135** by way of one or more application programming interfaces (APIs) (not shown). Because the operating system **134** incorporates these services, developers of applications programs **135** need not redevelop code to use the services. Examples of APIs provided by operating systems such as Microsoft's "WINDOWS" are well known in the art.

[0040] The computer **110** may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, **FIG. 1** illustrates a hard disk interface **140** that reads from and writes to non-removable, nonvolatile magnetic media, a magnetic disk drive **151**, which may be internal or external, that reads from and writes to a removable, nonvolatile magnetic disk **152**, and an optical disk drive **155** that reads from and writes to a removable, nonvolatile optical disk **156** such as a CD ROM. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, DVDs, digital video tape, solid state RAM, and solid state ROM. The hard disk drive **141**, which may be internal or external, is typically connected to the system bus **121** through a non-removable memory interface such as interface **140**, and magnetic disk drive **151** and optical disk drive **155** are typically connected to the system bus **121** by a removable memory interface, such as interface **150**.

[0041] The drives and their associated computer storage media discussed above and illustrated in **FIG. 1** provide storage of computer-readable instructions, data structures, program modules, and other data for the computer **110**. In **FIG. 1**, for example, hard disk drive **141** is illustrated as storing an operating system **144**, application programs **145**, other program modules **146**, and program data **147**. Note that these components can either be the same as or different from the operating system **134**, application programs **135**, other program modules **136**, and program data **137**. The operating system **144**, application programs **145**, other program modules **146**, and program data **147** are given different numbers here to illustrate that they may be different copies. A user may enter commands and information into the computer **110** through input devices such as a keyboard **162** and pointing device **161**, commonly referred to as a mouse, trackball, or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, and scanner. These and other input devices are often connected to the processing unit **120** through a user input interface **160** that is coupled to the system bus, but may be connected by other interface and bus structures, such as a parallel port, game port, or a universal serial bus (USB). A monitor **191** or other type of display device is also connected to the system bus **121** via an interface, such as a video interface **190**. In addition to the monitor, computers may also include other peripheral output devices such as speakers **197** and printer **196**, which may be connected through an output peripheral interface **195**.

[0042] The computer **110** may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer **180**. The remote computer **180** may be a personal computer, a server, a router, a network PC, a peer device, or other common network node, and typically includes many or all of the elements described above relative to the computer **110**, although only a memory storage device **181** has been illustrated in **FIG. 1**. The logical connections depicted in **FIG. 1** include a local area network (LAN) **171** and a wide area network (WAN) **173**, but may also include other networks. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets, and the Internet.

[0043] When used in a LAN networking environment, the computer **110** is connected to the LAN **171** through a network interface or adapter **170**. When used in a WAN networking environment, the computer **110** typically includes a modem **172** or other means for establishing communications over the WAN **173**, such as the Internet. The modem **172**, which may be internal or external, may be connected to the system bus **121** via the user-input interface **160**, or via another appropriate mechanism. In a networked environment, program modules depicted relative to the computer **110**, or portions thereof, may be stored in a remote memory storage device. By way of example, and not limitation, **FIG. 1** illustrates remote application programs **185** as residing on memory device **181**, which may be internal or external to the remote computer **180**. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

[0044] In the description that follows, the invention will be described with reference to acts and symbolic representations of operations that are performed by one or more computers, unless indicated otherwise. As such, it will be understood that such acts and operations, which are at times

referred to as being computer-executed, include the manipulation by the processing unit of the computer of electrical signals representing data in a structured form. This manipulation transforms the data or maintains them at locations in the memory system of the computer, which reconfigures or otherwise alters the operation of the computer in a manner well understood by those skilled in the art. The data structures where data are maintained are physical locations of the memory that have particular properties defined by the format of the data. However, while the invention is being described in the foregoing context, it is not meant to be limiting as those of skill in the art will appreciate that various of the acts and operations described hereinafter may also be implemented in hardware.

[0045] An embodiment of the present invention illustrated in **FIG. 2** enables a software developer to efficiently develop applications suitable for particular applications, including those for accessing, managing and otherwise utilizing databases. During step **200** a programmer or developer generates program code comprising procedural and non-procedural programming languages. Examples of suitable programming languages include ESQL that allows embedding SQL commands in programs written in the C programming language. The program code is pre-processed by a macro-preprocessor during step **205**. During step **205** the macro-preprocessor examines the source code for an occurrence of one or more specified statements. During step **210** any one of the specified statements is detected. Then the macro-preprocessor introduces a declaration for a first variable that is explicitly not defined. In the case of ESQL the first variable is declared using the "extern" key word to indicate to a compiler that no storage should be allocated since it would be allocated in another module. The macro-preprocessor also inserts statements that are in the form of function calls compliant with the Open Data Base Connectivity ("ODBC") standard during step **215**, but may include other mechanisms. Step **215** provides the proprietary implementations for a particular database since, as explained earlier, the actual implementation of a non-procedural language statement is not specified.

[0046] During step **220** a compiler converts the macro-preprocessor output to low-level instructions. Next, during step **225** a linker starts to resolve references using a library having a first object module that includes a definition for the first variable introduced by the macro-preprocessor at step **210**. If the linker detects during step **230** that the variable introduced by the macro-preprocessor has to be resolved then control transfers to step **235**. During step **235** the linker links the first object module having a definition for the declared variable introduced by the macro-preprocessor from the appropriate library. On the other hand, during step **230** if the linker does not detect the variable introduced by the macro-preprocessor then control is transferred to step **240**. During step **240** the linker does not link in the first object module since the first variable does not need to be resolved. As is readily apparent, the presence or absence of the first module in the executable is strictly dependent on the presence or absence of the specified statement tested in step **205**.

[0047] **FIG. 3** illustrates an embodiment of the invention enabling setting the value of a variable to a non-default value in response to detecting a specified statement. Steps **300** and **305** of **FIG. 3** correspond to steps **225** and **230** respectively of **FIG. 2**. If the declaration introduced by the macro-preprocessor does not have a definition, then control passes from step **305** to step **310**. During step **310** the linker includes a first object module to resolve the first variable. Furthermore, the first object module introduces a non-default value for a second variable.

[0048] If the linker does not encounter a declaration for the first variable, i.e., the macro-preprocessor did not detect the specified statement, then there is no first variable to resolve and control passes to step **315** from step **305**. During step **315**, the linker bypasses the first module because the first variable does not have to be resolved. During step **320** the linker includes a second object module to carry out a command, such as a DELETE command, in the source code. If the second object module includes a declaration for the second variable without a definition, as determined in step **325**, then the control shifts to step **330**.

[0049] The determination of the second variable declaration during step **325** results from the second variable's presence in a symbol table for the second object module and the absence of a corresponding memory allocation. During step **330** the linker continues to scan the linker libraries in an effort to resolve the second variable. Step **335** includes inclusion of a third module by the linker to resolve the second variable. The third object module, that is also the last module in the linker library, provides a definition setting a default value, e.g., 0, for the second variable. This value is in contrast with the non-default value set in the first object module. The linker did not include the first module since the first object module was encountered prior to the entry of the unresolved reference to the second variable included during step **325**.

[0050] It should be noted that while the linker encounters the first module prior to the second module, the third object module is encountered after the second module. On the other hand, during step **325** if the linker does not detect a second variable to be resolved, then control passes to step **340**. During step **340** the linker continues without resolving the second variable or including the third object module.

[0051] A software application includes one or more object modules that often correspond to source code files as is shown in **FIG. 4**. A software program **400** has one or more source files **405** corresponding to the object modules. Some of the source files include commands in a higher-level language, such as function calls or even instructions in scripted language. Code corresponding to each of these commands is substituted to actually implement the instruction. Thus, a macro-preprocessor **415** converts the computer program **400** having files **405** in **FIG. 4** to yield preprocessed program code comprising files **410** and possibly additional statements **420** in a compilable language.

[0052] An example of such a system is the SQL language and its extension in ESQL. SQL ensures that some standardized tasks can be performed without locking users into particular implementations. ESQL enables embedding SQL statements in one or more higher-level languages. A macro-preprocessor replaces the embedded SQL statements by implementation-specific code compiled along with the higher-level language statements. Thus, an exemplary ESQL processor works by reading C language statements with interspersed Structured Query Language (SQL) statements. The SQL statements are converted into Open Database

Connectivity (ODBC) compliant calls. The resulting source code is compiled and linked. For illustration purposes, FIG. 4 shows the result of compiling the preprocessed program code **420** by compiler **425** to generate object code. This object code includes, in the various object modules contained therein, unresolved references **430**, compiled code **435** corresponding to the files **410** and the compiled macro-preprocessor introduced statements **440**.

[0053] This object code is subsequently, acted upon by a linker **455**. Linker **455** supplies additional object modules to resolve unresolved references **430** by providing object code from libraries **445** and additional object files **450** that are included by the user. The linker **455** also ensures that the different object modules have the proper offsets relative to each other to allow execution of a single executable **460**. The executable **460** is executed in an environment similar to computing environment **100**.

[0054] In an embodiment of the present invention, a first statement, such as SAVEPOINT, in a program requires a different implementation for a second statement such as DELETE. Two possible implementations for the second statement independently designate performance characteristics at runtime. These implementations are provided in different object modules corresponding to the same instruction or statement. Thus, it is desirable that the code that actually implements the additional program instructions including the second statement should be sensitive to the occurrence or non-occurrence of the first statement.

[0055] Instead of requiring developers to examine all source code files to discover an occurrence of the first statement, a macro-preprocessor discovers an occurrence of the first statement. The macro-preprocessor is designed to respond to a context defined by the occurrence or non-occurrence of one or more statements of interest. Furthermore, modified linker libraries include object modules for resolving variables introduced by the macro-preprocessor. Additional modifications to the linker libraries allow object modules in the linker libraries to use one or more of the global variables representing a context while selecting code for execution at runtime.

[0056] Exemplary embodiments in accordance with the invention are described herein below without intending to limit the invention to these embodiments. In an embodiment of the invention the fact that a variable that is declared but not defined is set to default value, e.g., 0, is used to select code for execution. Thus, if the variable is given a non-default value upon encountering a statement of interest then code relevant to the statement of interest is executed. Selecting code based on the value of the variable results in faster code although without reduction in the footprint of the executable. The following pseudo-code illustrates such a variable to conditionally execute a code segment:

[0057] IF (_FIRST_STATEMENT_DETECTED == 0) THEN

[0058] {Execute efficient code for implementing second statement because the first statement is not being used}

[0059] ELSE

[0060] {Execute the code with the overhead for implementing the second statement because the first statement was detected}

[0061] END

[0062] The variable _FIRST_STATEMENT_DETECTED is declared and defined in a statement introduced by the macro-preprocessor if the macro-preprocessor encounters the first statement in any of the program files.

[0063] In the context of ESQL the implementation of the SQL statement, SAVEPOINT, provides an illustration of a global variable to flag a particular context. SAVEPOINT allows restoration of an earlier state, i.e., undoing a set of operations on a database. Therefore, if SAVEPOINT is used then the various state defining parameters need to be saved as other commands/statements are executed. Upon detecting SAVEPOINT the ESQL processor injects a declaration into the C stream of the form:

[0064] extern int_OCC_SAVEPOINT_USED;

[0065] The "extern" keyword informs the compiler that storage for _OCC_SAVEPOINT_USED is allocated in another file. Therefore, the compiler does not initialize _OCC_SAVEPOINT_USED. The linker uses two or more libraries such that the first library used by the linker contains as its first object module compiled code corresponding to the source code:

[0066] int_OCC_SAVEPOINT_USED = 1;

[0067] int _OCC_SAVEPOINT_ENABLED = 1;

[0068] and the second library contains in its last object module compiled code corresponding to the source code:

[0069] int _OCC_SAVEPOINT_ENABLED = 0;

[0070] In the linking process, if the macro-preprocessor injects a declaration for variable _OCC_SAVEPOINT_USED, the linker includes the first object module to provide a definition. The first object module provides a declaration and a value for _OCC_SAVEPOINT_ENABLED as illustrated. Subsequently, other object modules in the library include instructions that test variable _OCC_SAVEPOINT_ENABLED to flag whether SAVEPOINT has been used in any of the source files. If _OCC_SAVEPOINT_ENABLED is set, processing for SAVEPOINT will proceed. If _OCC_SAVEPOINT_USED is not set, the other object modules will not incur processing for SAVEPOINT.

[0071] In another embodiment of the invention, any program object module using the SAVEPOINT feature includes the declaration:

[0072] int_OCC_SAVEPOINT_NOT_USED;

[0073] A variable declaration assumes that upon first encountering the variable the compiler sets the variable to zero by default unless the contrary is indicated. However, this is not a requirement. And, multiple declarations in other object modules are harmless. The linker uses a library containing an object module having compiled code corresponding to the following code:

[0074] int_OCC_SAVEPOINT_NOT_USED = 1;

[0075] The macro-preprocessor declares _OCC_SAVEPOINT_NOT_USED resulting in the linker including the library object module setting _OCC_SAVEPOINT_NOT_USED to 1 only if _OCC_SAVEPOINT_NOT_USED is not declared elsewhere in the main program. (Note carefully the logical NOT: if the variable is not used in the main

program, it is set to one or TRUE.) Then any other object module in the library can test _OCC_SAVEPOINT_NOT_USED to decide if SAVEPOINT is used in any module in the program.

[0076] In another embodiment of the invention, a program object module using the SAVEPOINT feature will include a series of declarations having a global scope:

[0077] int _OCC_SAVEPOINT_USED;

[0078] The first object module in the library contains compiled code corresponding to

[0079] int_OCC_SAVEPOINT_USED = 1;

[0080] int_OCC_SAVEPOINT_MODULE_1;

[0081] int_OCC_SAVEPOINT_MODULE_2;

[0082] . . . and so on up to the number of separate object modules containing code dedicated for implementing SAVEPOINT. Each of the separate object modules for implementing SAVEPOINT contain a matching definition

[0083] int_OCC_SAVEPOINT_MODULE_1 = 1;

[0084] The first object module is included in the executable program [Please clarify the reason for the linker including the first object module since _OCC_SAVEPOINT_USED is presumably set to 0 by the compiler and does not need to be resolved by the linker. The same reason will presumably apply to the remaining object modules. If not, please clarify.]. In this way, those object modules containing code for SAVEPOINT can be included in the executable when required.

[0085] The invention uses well known rules for preprocessing, compiling and linking computer programs, particularly programs using the C or C++ programming language to improve the development of application programs. It provides a method for developing a computer program using a non-procedural programming language. The method includes declaring at least one first variable in a first source file responsive to detecting a first statement conforming to the non-procedural programming language. A macro-preprocessor examining the source code introduces a declaration statement for the first variable. A compiler compiles the first source file to generate a first object module followed by linking using at least one library. The linker includes a second object module containing a definition for the first variable to resolve the first variable. The second object module includes access to code to support implementation of the first statement. This access includes references to functions that are invoked by other statements to ensure proper execution of the first statement.

[0086] Furthermore, an additional non-procedural programming language is used to provide a third statement conforming to the additional non-procedural programming language.

[0087] The flowchart in **FIG. 5** describes an embodiment of the invention that enables including object modules from linker libraries to support additional overhead in the implementation of a DELETE statement. The additional overhead is required by the execution of one or more additional specified statements such as SAVEPOINT. In step **500** a computer program having the DELETE non-procedural programming language statement is pre-processed. However, the additional buffering overhead needs to be incurred only if the SAVEPOINT command or the ROLLBACK command is used in the computer program.

[0088] To flag the need for incurring an overhead the pre-processor introduces a declaration for a variable _VAR1 as being an "extern" upon encountering a ROLLBACK or SAVEPOINT statement during step 505. Declaring the variable to be "extern," in a C like programming language, informs the compiler that the variable definition is in another file external to the file being compiled. Consequently, the compiler does not initialize the variable during compilation in step **510**.

[0089] Following compilation, the object files generated by the compiler during step **510** are linked in step **515** using a linker program that resolves variable references in the object files. The linker detects if _VAR1 lacks a definition during step **520**. If _VAR1 lacks a definition then the linker resolves _VAR1 by examining the linker libraries for an object module having a definition for _VAR1, i.e., specifying memory for _VAR1. A value for a variable can be specified only after memory allocation for storing the variable value. During step **525** the linker encounters a first linker library V_1 having an object module M_1 that provides a definition setting _VAR1 to 1 along with a definition setting an additional global variable _VAR2 to a non-default value of 1. During step **525** inclusion of object module M_1 by the linker results in automatically including _VAR2 in a symbol table for the program being created by the linker. Other object modules included by the linker can include instructions to test the value of _VAR2 to detect if object module M_1 has been included. Notably, _VAR2 does not occur in the computer program, but is used in one or more of the additional object modules included in the linker libraries.

[0090] In the absence of an unresolved occurrence of _VAR1 in the pre-processed code, the linker does not include object module M_1. Instead control is transferred to step **530**. During step **530**, the occurrence of _VAR2 in other object modules included by the linker results in the linker including an object module M_2 to provide a declaration for _VAR2. The object module M_2 and M_1 can be given the same name but are not in the same linker library, V_1. Thus, the linker includes only one of modules M_1 or M_2. In other words, the linker includes object module M_1 prior to encountering object module M_2 if a SAVEPOINT or ROLLBACK command is encountered. Otherwise, object module M_2 is included, thus precluding any need to include object module M_1 (step **525**). The executable constructed by this procedure differs in its size based on whether SAVEPOINT or ROLLBACK commands are being used.

[0091] **FIG. 6** further illustrates an embodiment in accordance with the invention. In **FIG. 6** preprocessed program code **600** includes files such as file **605** and file **610** with file **605** having some statements **615** introduced by the preprocessor. Preprocessed code **600** is compiled to obtain object code **630** having object module, e.g., object module **620** and object module **625**. Linker links object modules using link libraries. The linked version includes conditional on an unresolved declaration of _VAR1 object module M_1 **640** resulting in a fat version **650**. The object module M_1 provides support for the extra overhead in response to the

macro-preprocessor declaration **615** in the preprocessed program code **600**. In contrast the absence of the macro-preprocessor declaration **615** results in the inclusion of object module M_2 **645** and generation of a thin version **655**.

[0092] FIG. **7** is a flowchart that tracks the implementation of a statement AA in a computer program in accordance with the invention. Statement AA has at least two possible code implementations that are suited to contexts defined by the presence or absence of another statement BB. If statement BB is present then implementation code A1 is preferred while the absence of statement BB results in implementation A2 being preferred. The preprocessing of the computer program begins at step **700**. During step **705** the macro-preprocessor examines source code to detect the presence of statement BB. If statement BB is present then the pre-processor introduces a declaration for a variable _VAR1 during step **710**. On the other hand, a failure to detect statement BB results in no such declaration being introduced during step **715**. The compiler compiles the pre-processor output during steps **720** and **725** following steps **710** and **715** respectively.

[0093] Following compilation, the linker does not locate a definition of _VAR1 because it does not occur elsewhere in the computer program. Consequently, during step **730**, which follows step **725**, the linker does not include the first object module M_1. Object module M_1 has a definition for _VAR1 and a declaration and definition setting another variable _VAR2 to a non-default value where _VAR2 is found in object modules in the linker libraries but not in the computer program. On the other hand, introduction of _VAR1 by the macro-preprocessor results in the inclusion of object module M_1 by the linker during step **735**, which follows step **720**, since the linker encounters the object module M_1 prior to the object module M_2. Note that object module M_2 is the last object module in each linker library used by the linker and introduces a default value, such as 0 for _VAR2.

[0094] Following inclusion of object module M_1 the linker does not include object module M_2. In contrast, since object module M_2 includes a definition of _VAR2, it is included if object module M_1 is not included to resolve references to _VAR2 in other object modules. Thus, the inclusion of M_1 and M_2 by the linker is on a mutually exclusive basis. M_1 provides access to code supporting added functionality required by the statement AA if statement BB occurs in the computer program. M_2, on the other hand, has no such functionality.

[0095] In an embodiment of the invention, an application uses a global variable to alter its behavior based on the use or non-use of a feature. Because the detection of the value of the global variable's value occurs at run-time, the code supporting both cases is linked into the application as illustrated below:

```
if (_FEATURE_XYZ_USED) {/* Or, if(!_FEATURE_XYZ_NOT_USED) */
do_thing_the_XYZ_way( );
    } else {      /* feature XYZ not used */
        do_thing_the_other_way( );
    }
```

[0096] The application is not linked against either library. Instead, statically linked wrappers with those function names are provided. These wrappers explicitly load the correct library and obtain the address of the requested function within that library. Thereafter, they merely forward all calls to the dynamically loaded function.

[0097] In order to produce an executable, the linker must include both do_thing_the_XYZ_way( ) and do_thing_the_other_way( ). In order to reduce the runtime footprint the implementations of do_thing_the_XYZ_way( ) and do_thing_the_other_way( ) are placed in separately named dynamically loaded libraries. The program itself makes calls to do_thing_the_XYZ_way( ) with that entry-point resolved at link-time from a library containing the wrapper described immediately above. The first time the wrapper is invoked at run-time, the wrapper loads the appropriate dynamically loaded library, finds and stores the entry-point of the same-name function within that library, and invoked that entry-point. At subsequent invocations of the wrapper, the stored entry-point is used immediately with no additional overhead.

[0098] In an exemplary embodiment of the invention, given that a stub routine do_thing( ) is statically linked into the executable, a dynamically-linked version of the code is as follows:

```
int first_call = 0;
    . . .
if( first_call) {
    first_call = 1;
    proper_routine = choose_thing( );
}
proper_routine( );
    . . .
where choose_thing( ) consists of the following code:
choose_thing( ) {
    if(_FEATURE_XYZ_USED) {
        open_dynamic_library(XYZ);
        return pointer to do_thing_the_XYZ_way( );
    } else {
        open_dynamic_library(not_XYZ);
        return point to do_thing_the_other_way( );
    }
```

[0099] In other words, in the first instance where invoking either do_thing_the_XYZ_way( ) or do_thing_the_other_way( ) results in the actual call being to the general choose_thing( ) function. In turn, choose_thing( ) checks the _FEATURE_XYZ_USED flag, opens the appropriate version of the dynamic library and loads the correct version of the routine. The choose_thing( ) routine returns the correct version, that overloads some other name, such as proper_routine( ). Then, upon calling the proper_routine( ) results in aliasing it to the correct version of the routine from the correct library.

[0100] In an alternative exemplary embodiment of the invention, very early in the executable program, the following code is executed:

```
main( ) {
    . . .
    if(_FEATURE_XYZ_USED ) {
        load_dl(do_XYZ_things);
    } else {
        load_dl(do_other_things);
    }
```

[0101] where load_dl( ) opens and loads a dynamic library. In this case, both the dynamically-loadable libraries do_XYZ_things and do_other_things contain the same entry points, except that in the first case they are written to use feature XYZ. After the appropriate dynamic library is loaded, the correct versions of the routines are used on subsequent calls.

[0102] In another embodiment of the invention, the wrapper includes a check on the global variable's state and reports an error if the application attempts to call the function in the wrong state. For instance, the do_thing_the_XYZ_way( ) would report an error if it found _FEATURE_XYZ_USED was false.

[0103] By avoiding the implicit link to both versions of the do_thing functionality (in separate libraries), "snapping" the links to the library with known entrypoints that are not going to be used based on link-time recognition is avoided with a reduction in the start-up overhead and memory consumption.

[0104] "Snapping" the links refers to run-time resolution of entry-points left unresolved at load-time. Entry-points flagged at load-time for resolution at run-time are stored in a special area of the program image file. When the program image file is loaded into memory at run-time, the entry-point linkages in this special area are filled with the correct addresses of the actual function entry-points in the dynamically loaded library containing the entry point linkages. This differs from load-time linking, where the address of the actual function is known at load-time. The operating system's program loader handles the "snapping" of links responsible for loading a program image into memory for execution.

[0105] Another embodiment indirectly calls the wrapper function through a function pointer-table entry. After the wrapper dynamically loads the appropriate library, it changes the function pointer-table entry to point directly to the matching function in the loaded library. The wrapper appears in a statically-linked library (i.e. its address is resolved at load-time). Following initialization of the above-described table, an entry "n" contains the address of the wrapper. If a program built in accordance with this embodiment wants to perform do_thing( ), then rather than directly invoking an entry-point (that is resolved at load time or snapped from some dynamic library at run time), the program invokes the function pointed to by entry "n" of the table. The first time this invocation occurs, the entry is pointing at the wrapper described above. This wrapper replaces the entry with a pointer to the correct function (i.e. the one indicated by the value of _FEATURE_XYZ_USED) and then calls that function. On subsequent invocations (invocation via entry "n" in the table), the correct function is called directly and without invoking the wrapper.

[0106] A "helper" function appears in the statically-linked library. This helper performs the functions of the aforementioned wrapper function for every entry in the table. The helper function checks the appropriate _FEATURE_xxxx-_USED variables, loads the matching library, finds the matching function name, and places the address of that entry-point into the appropriate table entry. If a program built in accordance with this embodiment wants to perform do_thing( ), then rather than directly invoking an entry-point (that is resolved at loadtime or snapped from some dynamic library at run time), the program invokes the function pointed to by entry "n" of the table. The first time any function available through this table is invoked (say, entry "j"), the entry is pointing at the wrapper described above. The wrapper invoked helper function replaces each entry in the table with a pointer to the correct function. The wrapper then calls the function via entry "i." On subsequent invocations (invocation via any entry "j" in the table), the correct function is called directly without the need to invoke the wrapper.

[0107] The preceding description illustrates the selection of different object modules by the linker to allow the same command to use two different implementations in a manner responsive to the program code environment. The invention is not limited to the embodiment described above or to the proposed implementation of the SAVEPOINT command. Other commands of interest can be similarly handled. The earlier illustrations of other ways for the macro-preprocessor to introduce statements are also easily adapted to result in the context-sensitive exclusion or inclusion of a particular object module.

[0108] Moreover, implementations of the invention include computer-readable medium having computer executable instructions for performing the steps of a method for constructing a computer program developed. The computer-readable medium has computer executable instructions for performing the step of declaring the first variable in the first source file by insertion of a first variable declaration statement by a macro-preprocessor responsive to the detection of the first statement of the non-procedural programming language.

[0109] The design of suitable linker libraries is modified in accordance with the invention. **FIG. 8** illustrates an embodiment of linker libraries with object modules and libraries corresponding to the order in which the libraries are used to resolve references. Naturally, first the unresolved references in the source program code are resolved followed by references that need to be resolved due to object modules so included. In **FIG. 8** a first linker library V_1 **800** has a first object module M_1 **805**. The first object module M_1 **805** has a definition for _VAR1 **810** and another variable _VAR2 with a non-default value **815**. The definition for _VAR2 implicitly introduces a declaration because memory allocation requires knowledge of the type for the variable. Another linker library V_m **825** has object modules M_1 **830**. In particular, linker library V_m **825** has an object module M_p **835** having a definition **840** setting _VAR2 to a default value.

[0110] Typically, the linker encounters the object module M_1 **805** earlier than any other object module, particularly

object module M_p **840**. Advantageously, object module M_p **840** is implemented as the last object module in a linker library to ensure the correct order of processing.

[0111] In view of the many possible embodiments to which the principles of this invention may be applied, it should be recognized that the embodiments described herein with respect to the drawing FIG.s is meant to be illustrative only and should not be taken as limiting the scope of invention. For example, those of skill in the art will recognize that the elements of the illustrated embodiment shown in software may be implemented in hardware and vice versa or that the illustrated embodiment can be modified in arrangement and detail without departing from the spirit of the invention. Therefore, the invention as described herein contemplates all such embodiments as may come within the scope of the following claims and equivalents thereof.

We claim:

1. A method of generating an executable from a computer program having a plurality of statements in a source code, the method comprising the steps of:

inserting, by a macro-preprocessor, a declaration statement for a first variable in a first source file in the computer program in response to detecting a first statement thereby rendering a modified first source file;

compiling the modified first source file to generate a first object module;

linking the first object module to generate an executable program, the linking step including resolving the first variable by linking a second object module having a definition corresponding to the first variable; and

providing a third object module in a linker library, the third object module corresponding to a second statement and including code supporting the first statement.

2. The method of claim 1, wherein the computer program includes statements conforming to a non-procedural programming language.

3. The method of claim 2, wherein the computer program is created using at least one statement conforming to an embedded structured query language.

4. The method of claim 1, wherein the first variable declaration excludes a definition for the first variable.

5. The method of claim 1, wherein the code to support the first statement in the third object module includes instructions to carry out additional tasks required by the first statement when implementing the second statement.

6. The method of claim 5, wherein the code to support the first statement is accessed in the third object module from the second object module.

7. The method of claim 2, wherein an additional non-procedural programming language is used to provide a third statement conforming to the additional non-procedural programming language, wherein in response to detecting the third statement a second variable is declared in at least one source file in the computer program.

8. The method of claim 1 wherein the macro-preprocessor does not declare the first variable in the modified first source file if the macro-preprocessor does not detect the first statement in the first source file whereby the first object module is not linked with the second object module.

9. The method of claim 7, wherein a fourth object module having a definition for the second variable is linked instead of the second object module.

10. A computer-readable medium having computer executable instructions for performing the steps of a method of generating an executable from a computer program having a plurality of statements in a source code, the method comprising the steps of:

inserting, by a macro-preprocessor, a declaration statement for a first variable in a first source file in the computer program in response to detecting a first statement thereby rendering a modified first source file;

compiling the modified first source file to generate a first object module;

linking the first object module to generate an executable program, the linking step including resolving the first variable by linking a second object module having a definition corresponding to the first variable; and

providing a third object module in a linker library, the third object module corresponding to a second statement and including code supporting the first statement.

11. A computer-readable medium as in claim 10, wherein the computer program includes statements conforming to a non-procedural programming language.

12. A computer-readable medium as in claim 11 having computer executable instructions in the computer program conforming to an embedded structured query language.

13. A computer-readable medium as in claim 10, wherein the code to support the first statement in the third object module includes instructions to carry out additional tasks required by the first statement when implementing the second statement.

14. A computer-readable medium as in claim 13, wherein the code to support the first statement is accessed in the third object module from the second object module.

15. A computer-readable medium as in claim 11, wherein an additional non-procedural programming language is used to provide a third statement conforming to the additional non-procedural programming language, wherein responsive to detection of the third statement a second variable is declared in at least one source file in the computer program.

16. A computer-readable medium as in claim 10, wherein the macro-preprocessor does not declare the first variable in the modified first source file if the macro-preprocessor does not detect the first statement in the first source file whereby the first object module is not linked with the second object module.

17. A computer-readable medium as in claim 15, wherein a fourth object module having a definition for the second variable is linked instead of the second object module.

18. A linker library for generating an executable by a linker linking compiled object modules, the linker library comprising a first object module having a definition for a first variable providing a first value to the first variable and a declaration and definition for a second variable providing a second value to the second variable whereby a second object module can test, at runtime, the second variable for the second value to determine if the first object module is linked by the linker into the executable.

19. The linker library of claim 18 wherein the linker library has only one object module.

20. The linker library of claim 18 further having a third object module after the first object module, the third object module having another declaration and definition for the second variable providing a third value to the second variable.

21. The linker library of claim 20 wherein the third object module is the last object module in the linker library.

22. A linker library comprising a last object module having a definition for a variable, the last object module following at least one object module in the linker library wherein the variable is declared in at least one object module distinct from the last object module.

23. A macro-preprocessor for preprocessing program source code, the program source code comprising programming language statements such that an implementation of a first statement requires changing an implementation of a second programming language, wherein the macro-preprocessor introduces a declaration for a first variable in a preprocessed program source code in response to detecting an occurrence of the first programming statement in the program source code.

24. The macro-processor of claim 23 wherein the programming language is a non-procedural programming language.

* * * * *

US005749079A

United States Patent [19]

Yong et al.

[11] Patent Number: 5,749,079

[45] Date of Patent: *May 5, 1998

[54] **END USER QUERY FACILITY INCLUDING A QUERY CONNECTIVITY DRIVER**

[75] Inventors: **Dennis Yong; Viktor Choong-Hung Cheng; Christopher Leng-Hong Yo; Liat Lim**, all of Singapore, Singapore

[73] Assignee: **Singapore Computer Systems Limited**, Singapore

[*] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,701,466.

[21] Appl. No.: **378,744**

[22] Filed: **Jan. 26, 1995**

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 346,507, Nov. 29, 1994, Pat. No. 5,701,466, which is a continuation-in-part of Ser. No. 154,343, Nov. 17, 1993, abandoned, which is a continuation-in-part of Ser. No. 846,522, Mar. 4, 1992, Pat. No. 5,325,465.

[51] **Int. Cl.**[6] **G06F 17/30**

[52] **U.S. Cl.** **707/100**; 395/700; 707/3

[58] **Field of Search** 395/600, 50, 54, 395/52, 63, 77, 51

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,631,664	12/1986	Bachman	364/200
4,815,005	3/1989	Oyanagi	395/63
4,982,340	1/1991	Oyanagi	395/63
5,043,929	8/1991	Kramer	364/578
5,072,406	12/1991	Ammon	395/64
5,123,103	6/1992	Ohtaki et al.	395/600
5,193,185	3/1993	Lanter	395/600
5,224,206	6/1993	Simoudis	395/77
5,283,856	2/1994	Gross et al.	395/51
5,325,465	6/1994	Hung et al.	395/63
5,386,571	1/1995	Kurz	395/700

FOREIGN PATENT DOCUMENTS

0 589 070 A1	3/1994	European Pat. Off.	G06F 15/40

OTHER PUBLICATIONS

Higa et al. "Object–Oriented Methodology for Knowledge base/Database coupling"; communications of the ACM. V35. n6. p99(15); Jun. 1992.

Tu et al. "Episodic Skeletal–plan refinement based on temporal data"; communications of the ACM vol. 32; pp. 1439–1451 Dec. 1989.

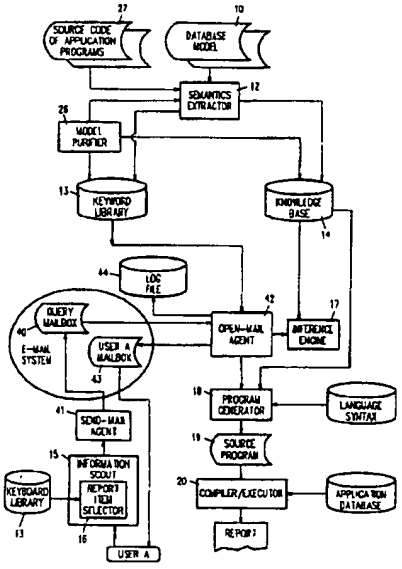
(List continued on next page.)

Primary Examiner—Thomas G. Black
Assistant Examiner—Jean G. Corrielus
Attorney, Agent, or Firm—Steven F. Caserza; Flehr Hohbach Test Albritton & Herbert LLP

[57] **ABSTRACT**

An end user query technology is taught which is capable of automatically understanding the database model and guiding the user to scout for the desired information, thereby increasing productivity and ease of information access. The user is freed from the need to understanding the database model, with the end user query facility of this invention quickly guiding the user to acquire the information. This is made possible by the end user query facility of this invention first recapturing the application semantics from the existing database model to provide a set of derived semantics. The derived semantics are then used by the end user query facility to intelligently guide the user to scout for the desired information in the database. In addition, the derived semantics can be easily updated by the end user query facility when the database model is changed.

82 Claims, 28 Drawing Sheets



OTHER PUBLICATIONS

Roger Jennings "Using Access for windows" pp. 1–7 Jan. 1989.

Russ Holden Digital's DB Integrator: A Commercial Multi–Database Management System (Digital Equipment Corporation) Sep. 1992.

Proceedings of the Third International Conference on Parallel and Distributed Information Systems (Cat. No. 94TH0668–4), Proceedings of 3rd International Conference on Parallel and Distributed Information Systems, Austin, TX, USA, 28–30 Sep. 1994, ISBN 0–8186–6400–2, 1994, Los Alamitos, CA, USA, IEEE Comput. Soc. Press, USA, pp. 267–268, Holden, R., "Digital's DB Integrator: A Commercial Multi–Database Management System".

Sixth International Conference on Data Engineering (Cat. No. 90CH2840–7), Los Angeles, CA, USA, 5–9 Feb. 1990, ISBN 0–8186–2025–0, 1990, Los Alamitos, CA, USA, IEEE Comput. Soc., USA, pp. 2–10, Kaul, M., et al., "ViewSystem: Integrating Heterogeneous Information Bases by Object–Oriented Views".

USENIX C++ Technical Conference Proceedings, Portland, OR, USA, 10–13 Aug. 1992, Berkeley, CA, USA, USENIX Assoc., USA, pp. 151–169, Alashqur, A., et al., "O–R Gateway: A System for Connecting C++ Application Programs and Relational Databases".

Journal of Information Science, 1994, UK, vol. 20, No. 4, ISSN 1352–7460, pp. 295–299, Sreenivasa, Ravi, et al. "An E–Mail–Based Bibliographic Information Server".

A Survey of the Universal Relation Model, Leymann, Data & Knowledge Engineering, vol. 4, 1989, pp. 305–320.

The Universal Relation as a user Interface, Ullinson, Principles of Database and Knowledge Based Systems, vol. II, Chpter 17, 1989.

Consequences of Assuming a Universal Relation, W. Kent, 1981 ACM Transactions on Database Systems, vol. 6, No. 4, pp. 539–556.

EasyTalk Product Backgrounder from Intelligent Business Systems (no date).

SQL Software Interfaces Brochure (no date).

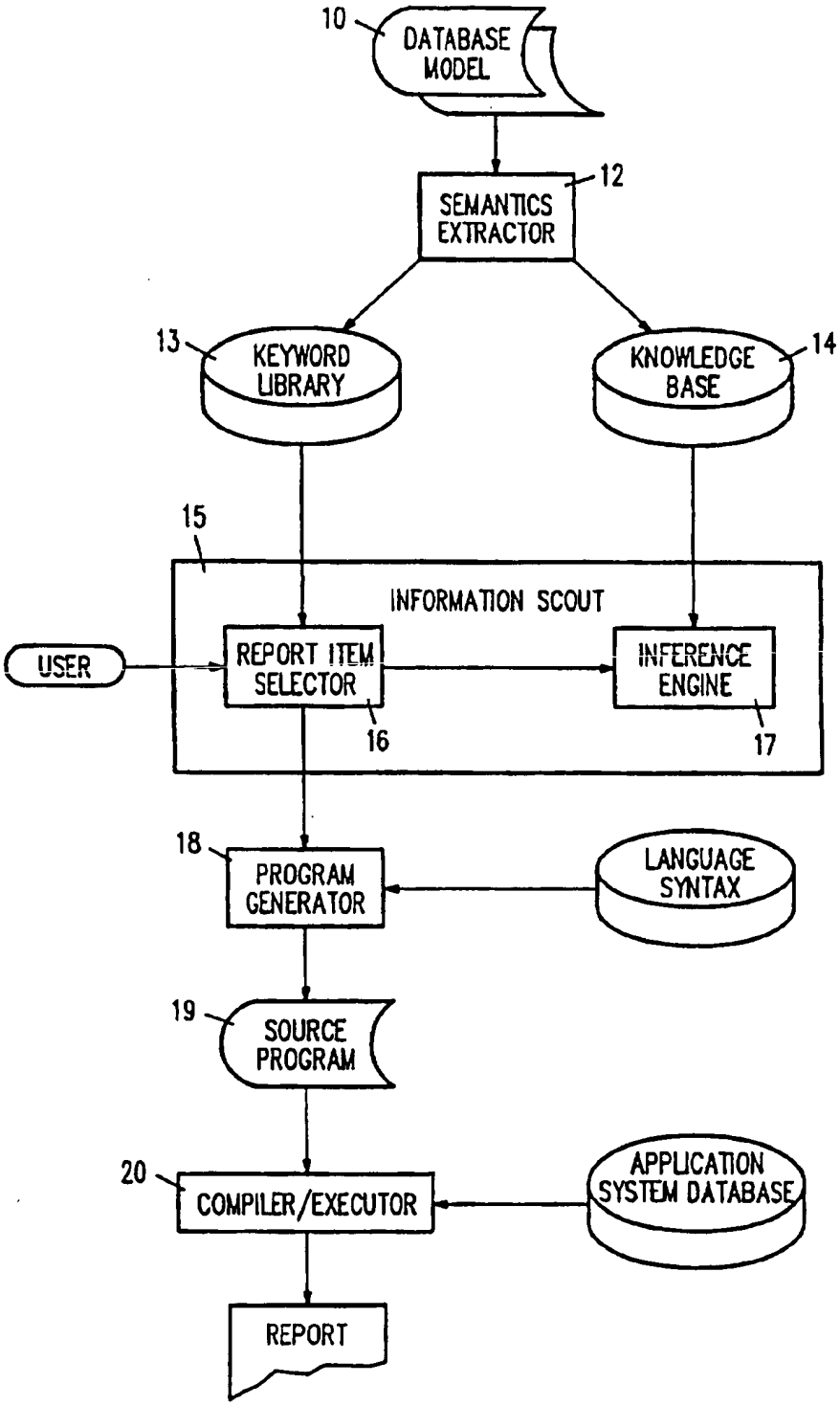

10
DATABASE MODEL
12
SEMANTICS EXTRACTOR
13
KEYWORD LIBRARY
14
KNOWLEDGE BASE
15
INFORMATION SCOUT
USER
REPORT ITEM SELECTOR
16
INFERENCE ENGINE
17
18
PROGRAM GENERATOR
LANGUAGE SYNTAX
19
SOURCE PROGRAM
20
COMPILER/EXECUTOR
APPLICATION SYSTEM DATABASE
REPORT

FIG. 1

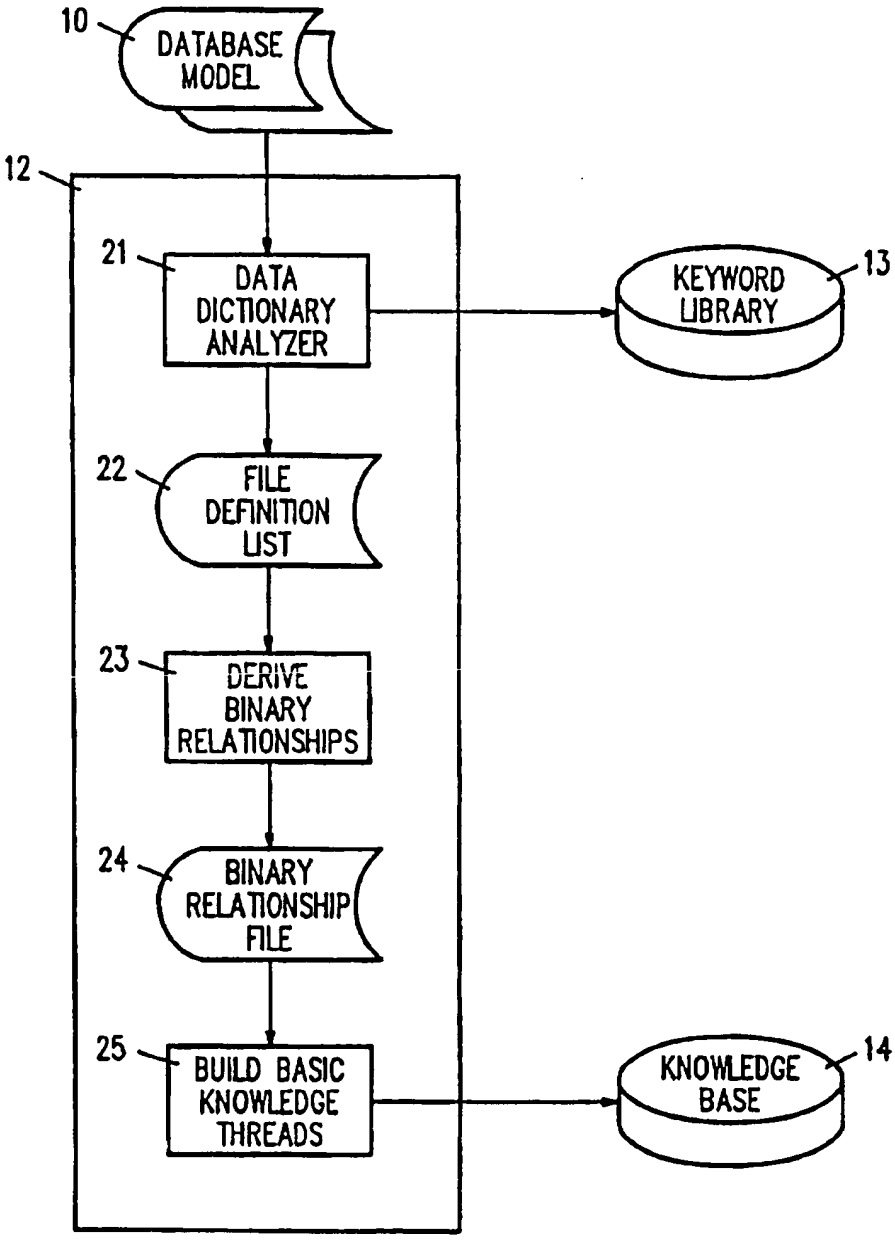

10
DATABASE MODEL
12
21
DATA DICTIONARY ANALYZER
KEYWORD LIBRARY
13
22
FILE DEFINITION LIST
23
DERIVE BINARY RELATIONSHIPS
24
BINARY RELATIONSHIP FILE
25
BUILD BASIC KNOWLEDGE THREADS
KNOWLEDGE BASE
14

FIG. 2

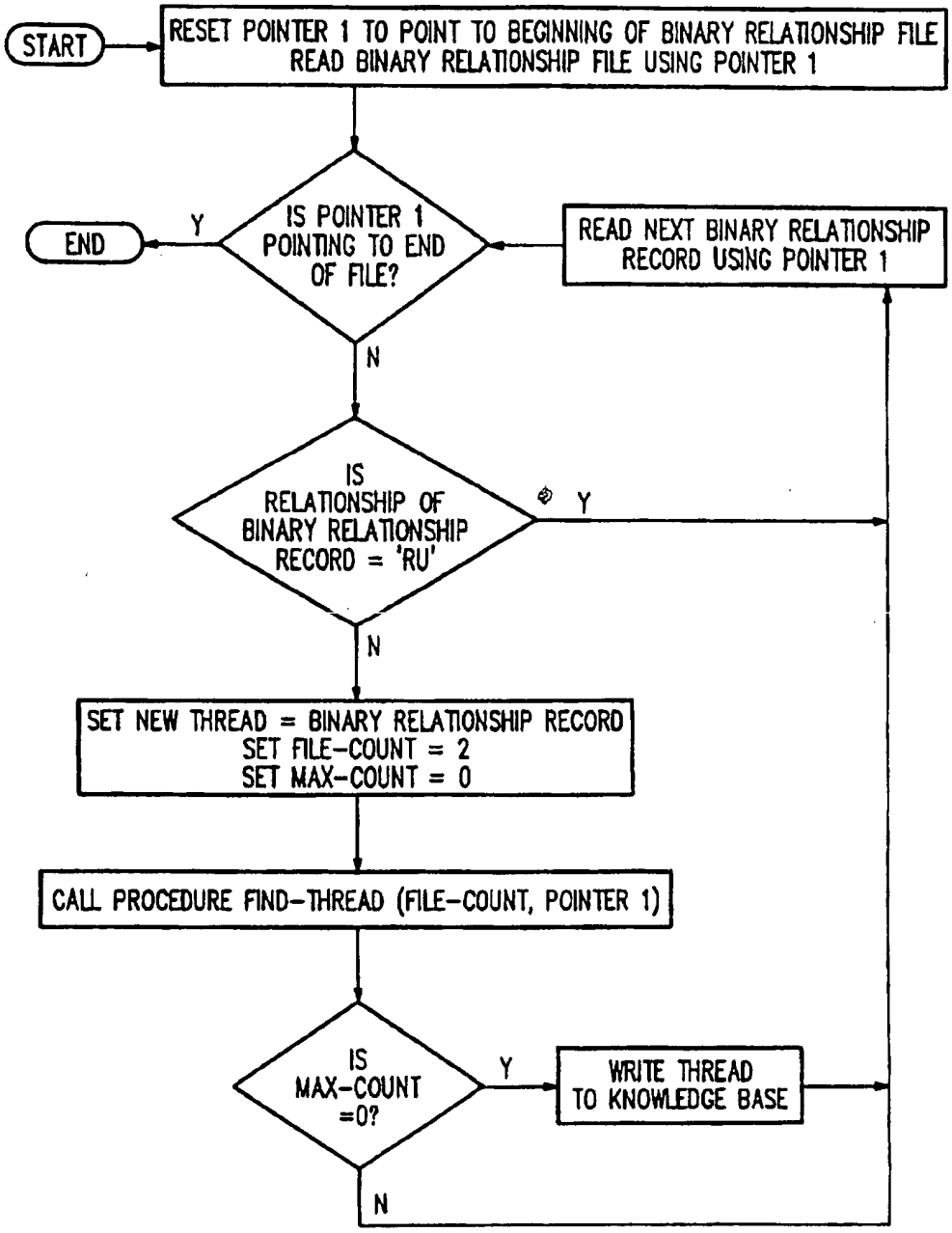

START
RESET POINTER 1 TO POINT TO BEGINNING OF BINARY RELATIONSHIP FILE
READ BINARY RELATIONSHIP FILE USING POINTER 1
IS POINTER 1 POINTING TO END OF FILE?
Y
END
READ NEXT BINARY RELATIONSHIP RECORD USING POINTER 1
N
IS RELATIONSHIP OF BINARY RELATIONSHIP RECORD = 'RU'
Y
N
SET NEW THREAD = BINARY RELATIONSHIP RECORD
SET FILE-COUNT = 2
SET MAX-COUNT = 0
CALL PROCEDURE FIND-THREAD (FILE-COUNT, POINTER 1)
IS MAX-COUNT =0?
Y
WRITE THREAD TO KNOWLEDGE BASE
N

FIG. 3

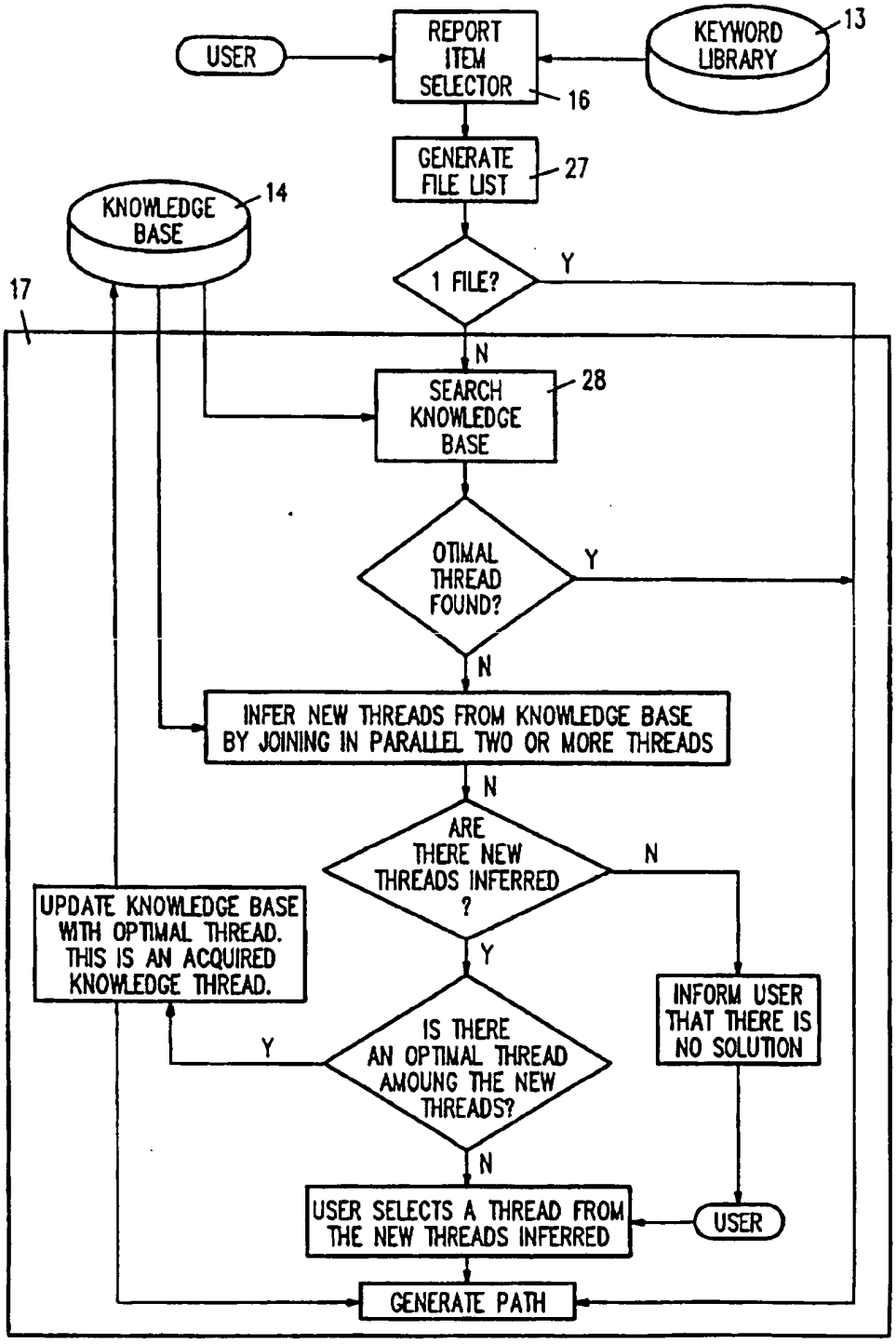

USER
REPORT ITEM SELECTOR
16
KEYWORD LIBRARY
13
GENERATE FILE LIST
27
KNOWLEDGE BASE
14
1 FILE?
Y
N
17
SEARCH KNOWLEDGE BASE
28
OTIMAL THREAD FOUND?
Y
N
INFER NEW THREADS FROM KNOWLEDGE BASE BY JOINING IN PARALLEL TWO OR MORE THREADS
N
ARE THERE NEW THREADS INFERRED ?
N
Y
UPDATE KNOWLEDGE BASE WITH OPTIMAL THREAD. THIS IS AN ACQUIRED KNOWLEDGE THREAD.
IS THERE AN OPTIMAL THREAD AMOUNG THE NEW THREADS?
Y
INFORM USER THAT THERE IS NO SOLUTION
N
USER SELECTS A THREAD FROM THE NEW THREADS INFERRED
USER
GENERATE PATH

FIG. 4

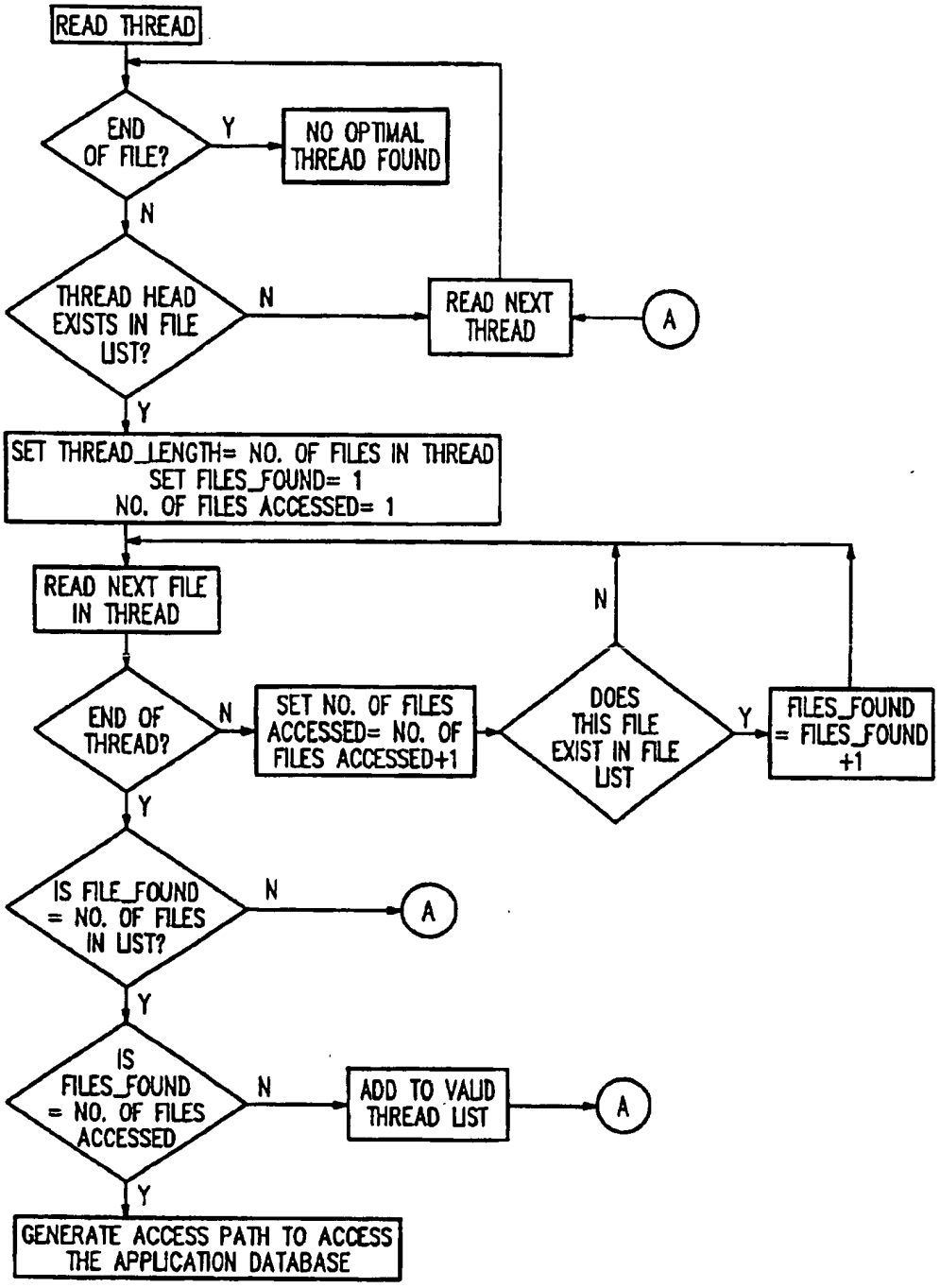

READ THREAD
END OF FILE?
Y
NO OPTIMAL THREAD FOUND
N
THREAD HEAD EXISTS IN FILE LIST?
N
READ NEXT THREAD
A
Y
SET THREAD_LENGTH= NO. OF FILES IN THREAD
SET FILES_FOUND= 1
NO. OF FILES ACCESSED= 1
READ NEXT FILE IN THREAD
END OF THREAD?
N
SET NO. OF FILES ACCESSED= NO. OF FILES ACCESSED+1
DOES THIS FILE EXIST IN FILE LIST
Y
FILES_FOUND = FILES_FOUND +1
N
Y
IS FILE_FOUND = NO. OF FILES IN LIST?
N
A
Y
IS FILES_FOUND = NO. OF FILES ACCESSED
N
ADD TO VALID THREAD LIST
A
Y
GENERATE ACCESS PATH TO ACCESS THE APPLICATION DATABASE

FIG. 5

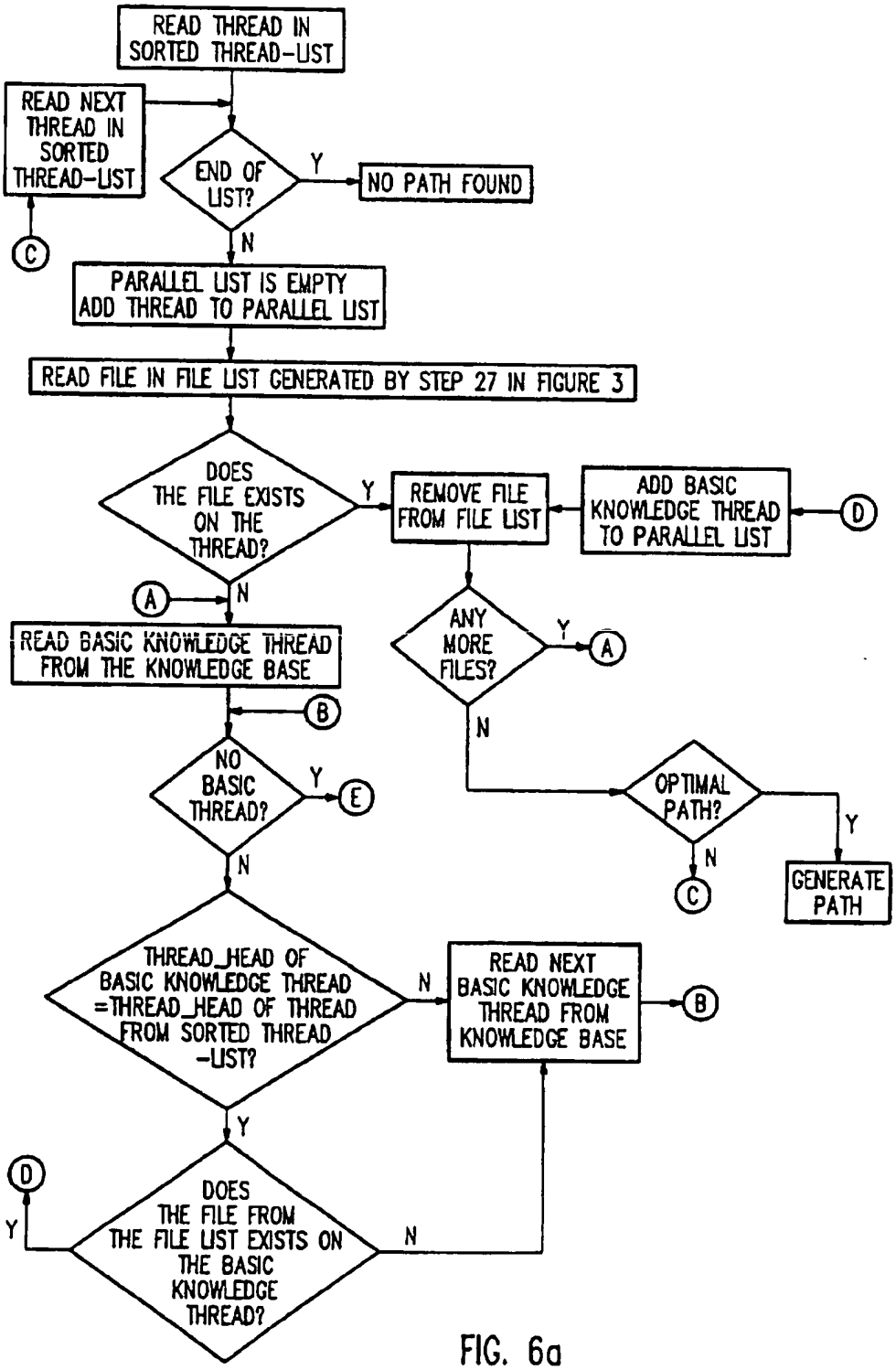

READ THREAD IN SORTED THREAD-LIST
READ NEXT THREAD IN SORTED THREAD-LIST
END OF LIST?
Y
NO PATH FOUND
N
C
PARALLEL LIST IS EMPTY ADD THREAD TO PARALLEL LIST
READ FILE IN FILE LIST GENERATED BY STEP 27 IN FIGURE 3
DOES THE FILE EXISTS ON THE THREAD?
Y
REMOVE FILE FROM FILE LIST
ADD BASIC KNOWLEDGE THREAD TO PARALLEL LIST
D
A
N
READ BASIC KNOWLEDGE THREAD FROM THE KNOWLEDGE BASE
ANY MORE FILES?
Y
A
N
B
NO BASIC THREAD?
Y
E
N
OPTIMAL PATH?
Y
N
C
GENERATE PATH
THREAD_HEAD OF BASIC KNOWLEDGE THREAD =THREAD_HEAD OF THREAD FROM SORTED THREAD -LIST?
N
READ NEXT BASIC KNOWLEDGE THREAD FROM KNOWLEDGE BASE
B
Y
D
Y
DOES THE FILE FROM THE FILE LIST EXISTS ON THE BASIC KNOWLEDGE THREAD?
N

FIG. 6a

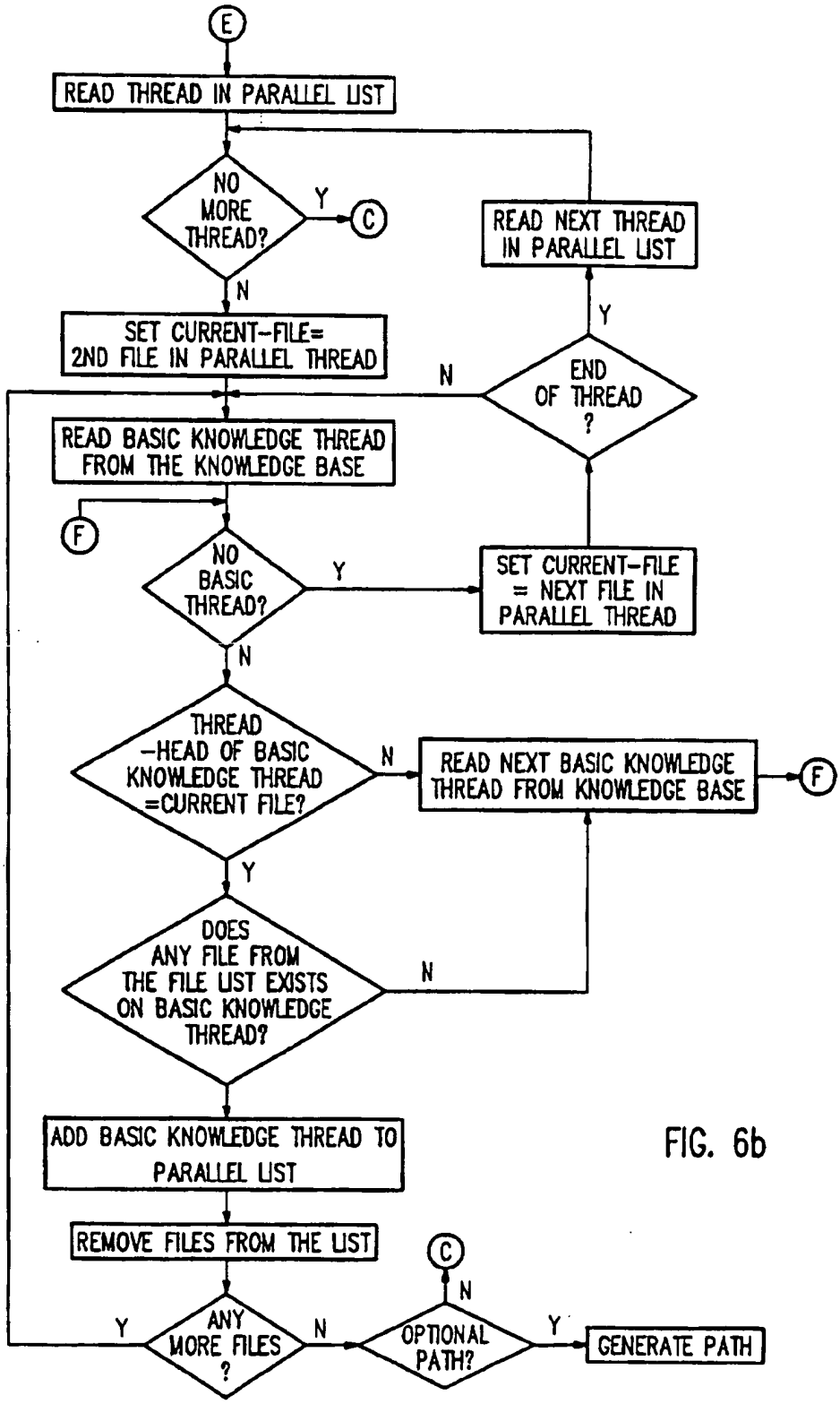

E
READ THREAD IN PARALLEL LIST
NO MORE THREAD?
Y
C
N
SET CURRENT-FILE= 2ND FILE IN PARALLEL THREAD
READ NEXT THREAD IN PARALLEL LIST
Y
END OF THREAD ?
N
READ BASIC KNOWLEDGE THREAD FROM THE KNOWLEDGE BASE
F
NO BASIC THREAD?
Y
SET CURRENT-FILE = NEXT FILE IN PARALLEL THREAD
N
THREAD -HEAD OF BASIC KNOWLEDGE THREAD =CURRENT FILE?
N
READ NEXT BASIC KNOWLEDGE THREAD FROM KNOWLEDGE BASE
F
Y
DOES ANY FILE FROM THE FILE LIST EXISTS ON BASIC KNOWLEDGE THREAD?
N
ADD BASIC KNOWLEDGE THREAD TO PARALLEL LIST
REMOVE FILES FROM THE LIST
C
N
Y
ANY MORE FILES ?
N
OPTIONAL PATH?
Y
GENERATE PATH

FIG. 6b

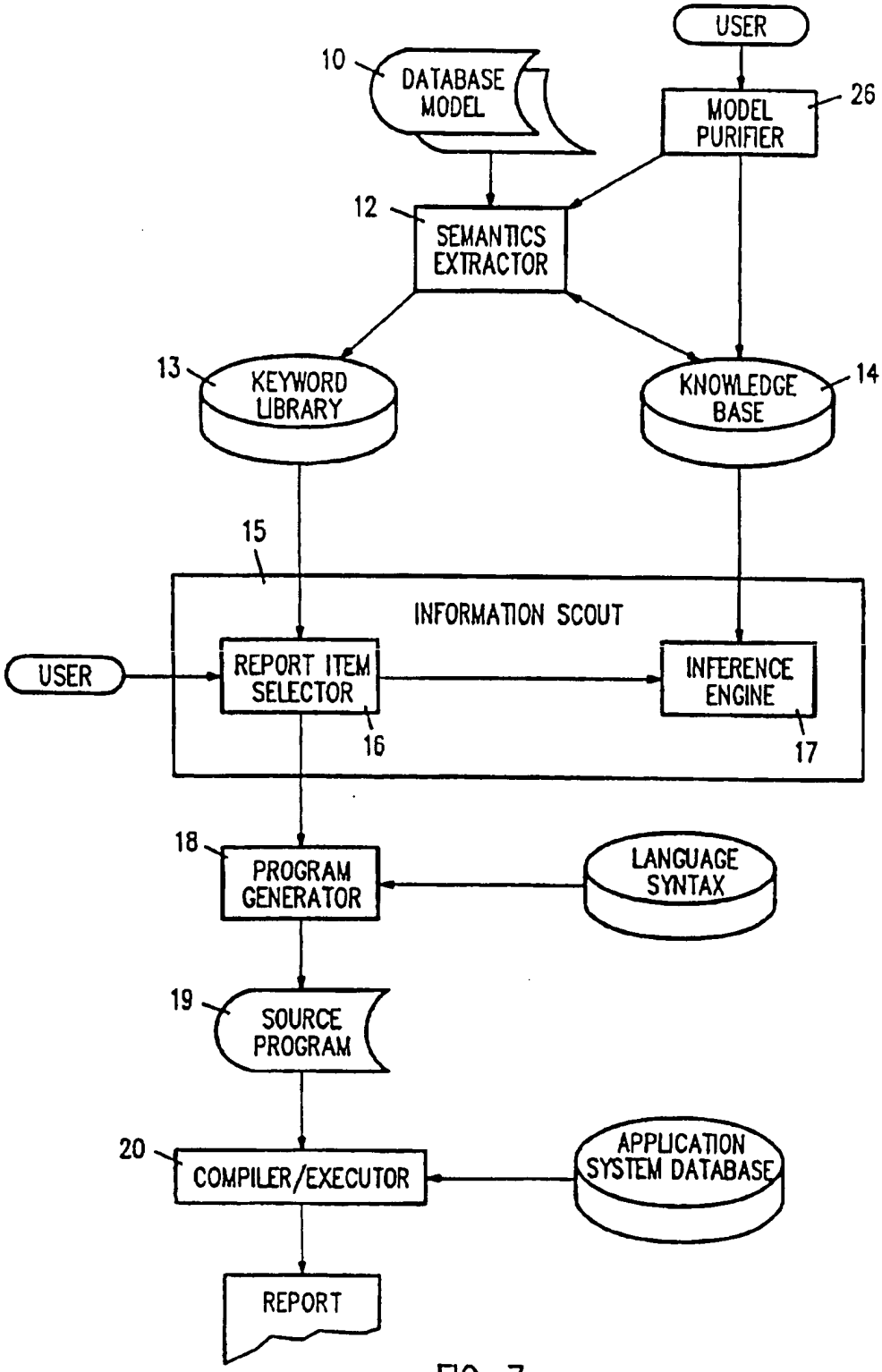

USER
10
DATABASE MODEL
MODEL PURIFIER
26
12
SEMANTICS EXTRACTOR
13
KEYWORD LIBRARY
KNOWLEDGE BASE
14
15
INFORMATION SCOUT
USER
REPORT ITEM SELECTOR
INFERENCE ENGINE
16
17
18
PROGRAM GENERATOR
LANGUAGE SYNTAX
19
SOURCE PROGRAM
20
COMPILER/EXECUTOR
APPLICATION SYSTEM DATABASE
REPORT

FIG. 7

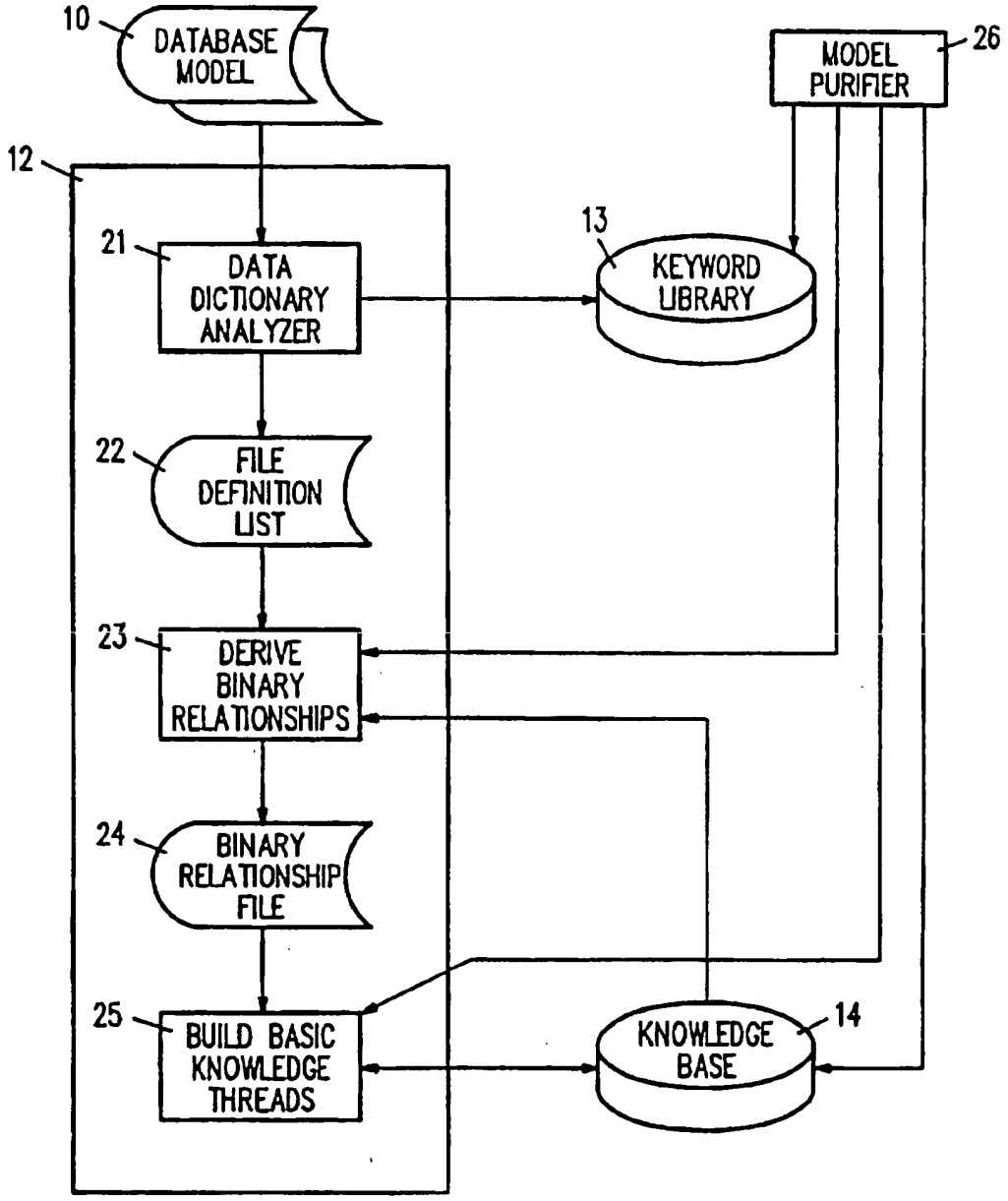

10
DATABASE MODEL
26
MODEL PURIFIER
12
13
21
DATA DICTIONARY ANALYZER
KEYWORD LIBRARY
22
FILE DEFINITION LIST
23
DERIVE BINARY RELATIONSHIPS
24
BINARY RELATIONSHIP FILE
25
BUILD BASIC KNOWLEDGE THREADS
14
KNOWLEDGE BASE

FIG. 8

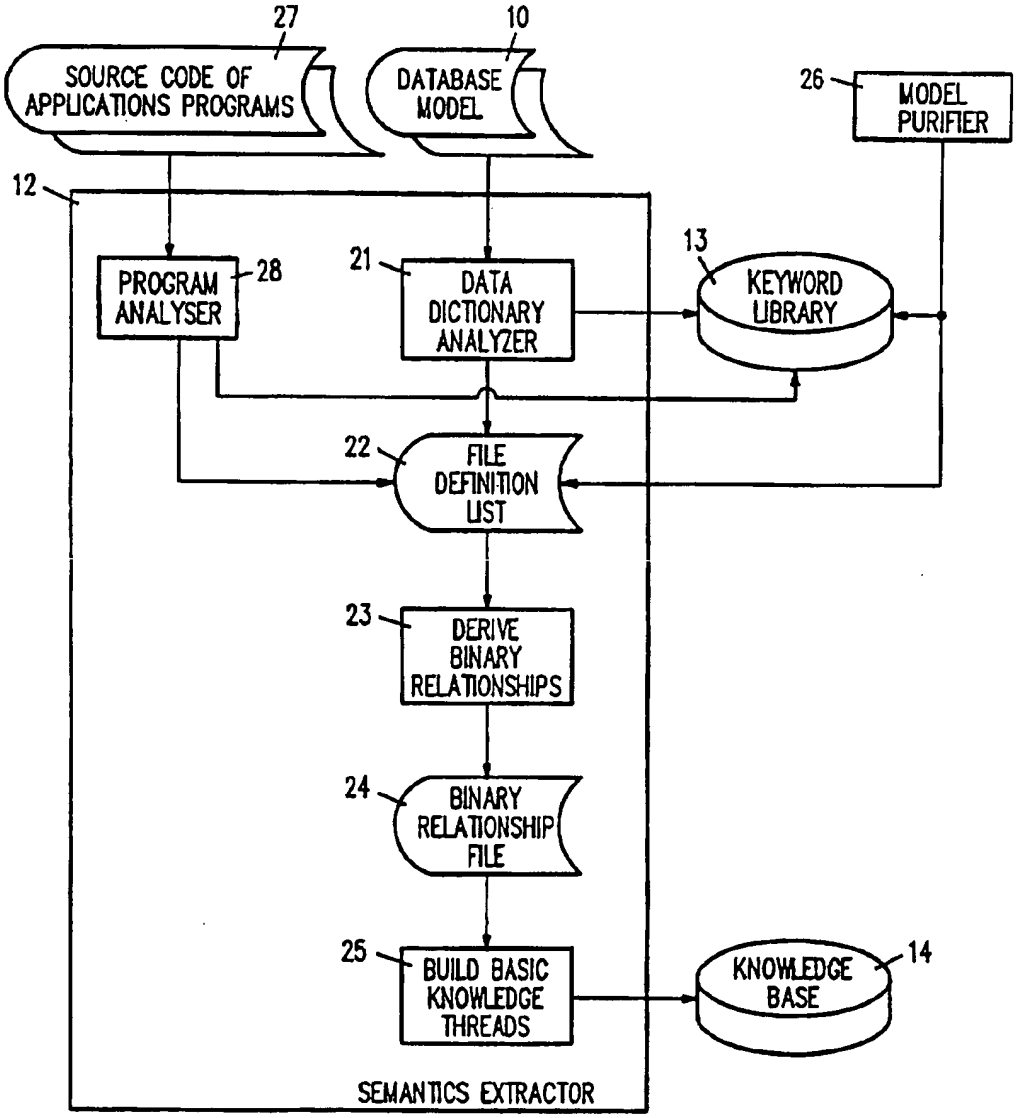

27
SOURCE CODE OF APPLICATIONS PROGRAMS
10
DATABASE MODEL
26
MODEL PURIFIER
12
PROGRAM ANALYSER
28
21
DATA DICTIONARY ANALYZER
13
KEYWORD LIBRARY
22
FILE DEFINITION LIST
23
DERIVE BINARY RELATIONSHIPS
24
BINARY RELATIONSHIP FILE
25
BUILD BASIC KNOWLEDGE THREADS
14
KNOWLEDGE BASE
SEMANTICS EXTRACTOR

FIG. 9

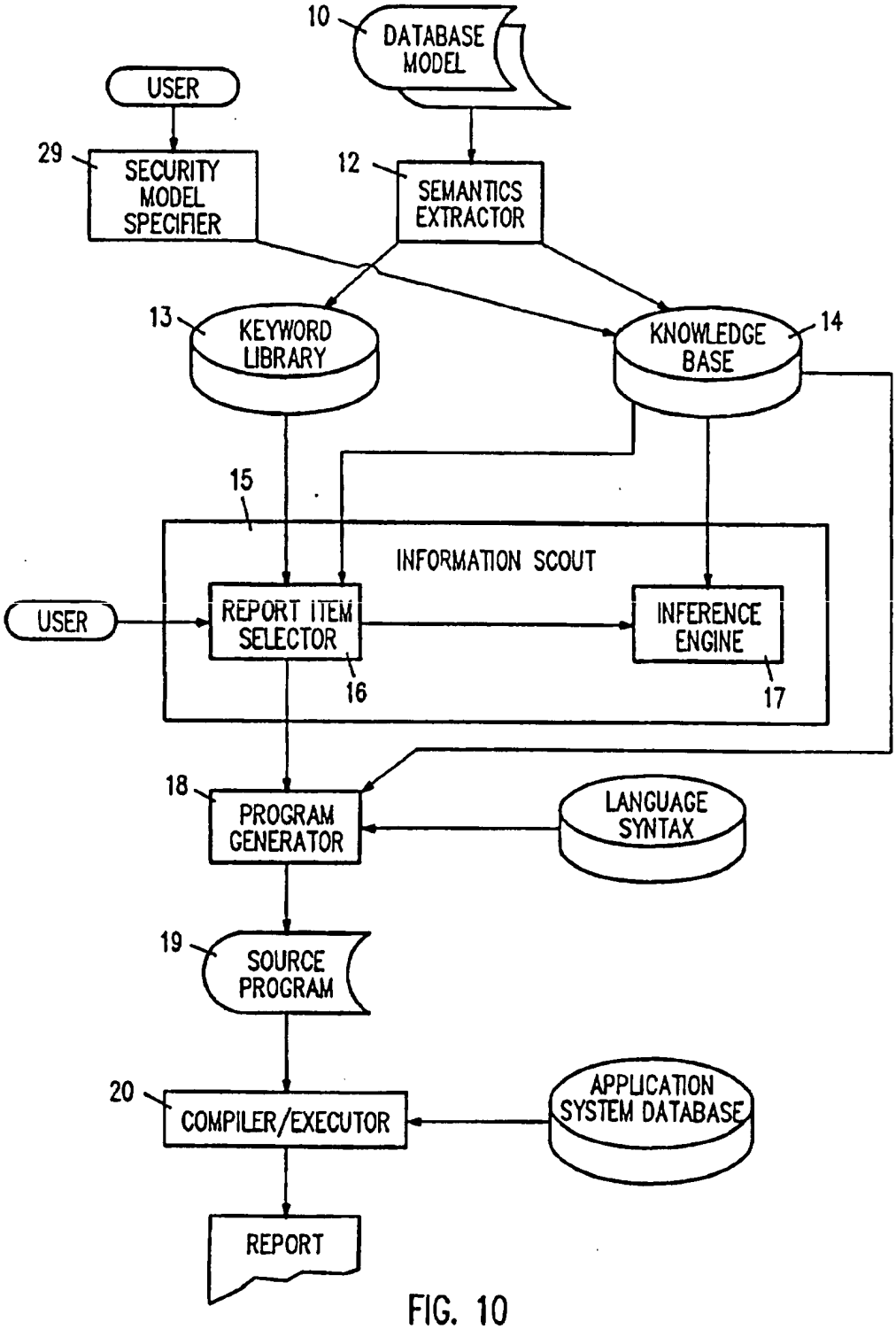

10
DATABASE MODEL
USER
29
SECURITY MODEL SPECIFIER
12
SEMANTICS EXTRACTOR
13
KEYWORD LIBRARY
KNOWLEDGE BASE
14
15
INFORMATION SCOUT
USER
REPORT ITEM SELECTOR
16
INFERENCE ENGINE
17
18
PROGRAM GENERATOR
LANGUAGE SYNTAX
19
SOURCE PROGRAM
20
COMPILER/EXECUTOR
APPLICATION SYSTEM DATABASE
REPORT

FIG. 10

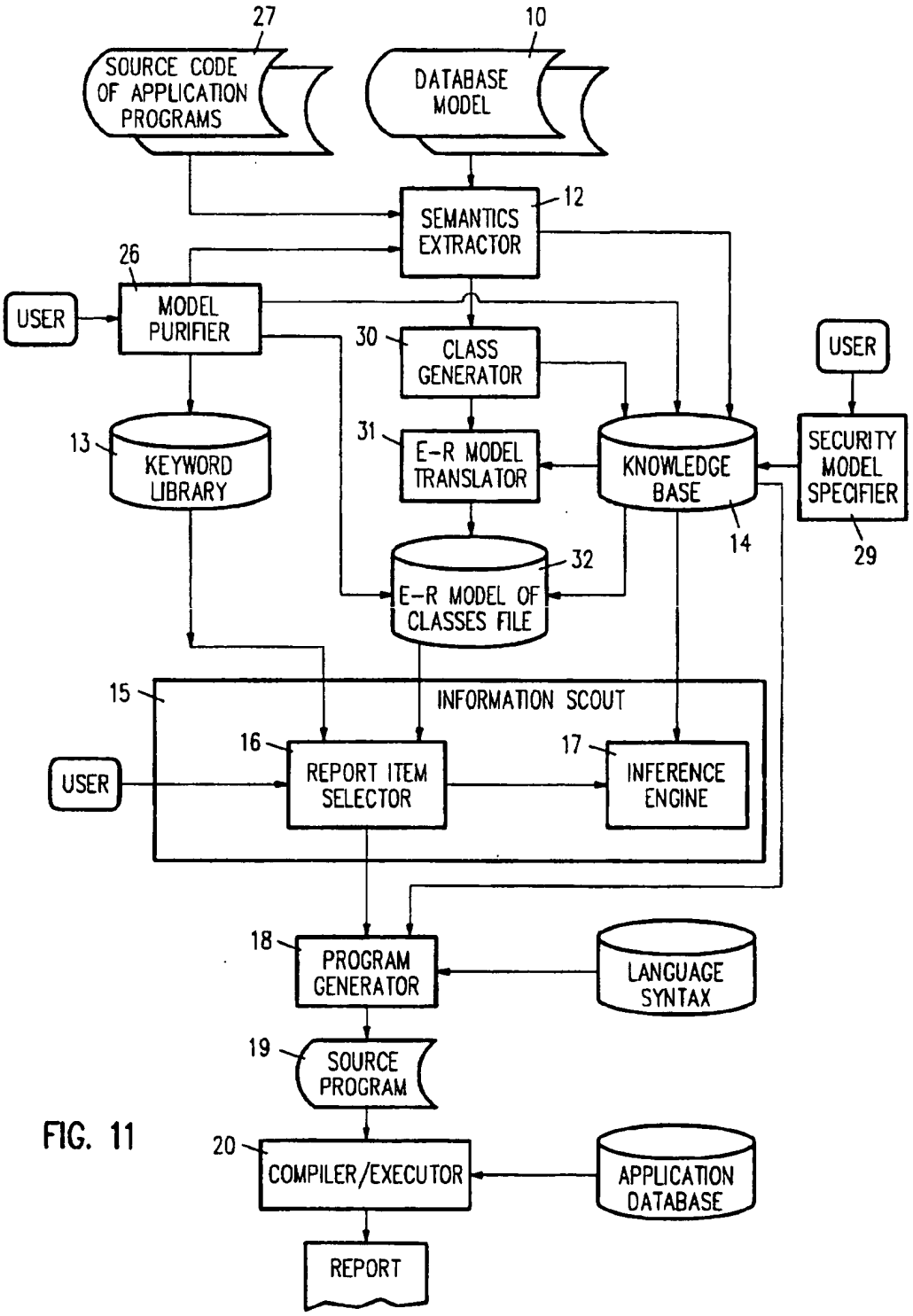

27
SOURCE CODE OF APPLICATION PROGRAMS
10
DATABASE MODEL
12
SEMANTICS EXTRACTOR
26
USER
MODEL PURIFIER
30
CLASS GENERATOR
USER
13
KEYWORD LIBRARY
31
E-R MODEL TRANSLATOR
KNOWLEDGE BASE
SECURITY MODEL SPECIFIER
14
29
32
E-R MODEL OF CLASSES FILE
15
INFORMATION SCOUT
16
17
USER
REPORT ITEM SELECTOR
INFERENCE ENGINE
18
PROGRAM GENERATOR
LANGUAGE SYNTAX
19
SOURCE PROGRAM
20
COMPILER/EXECUTOR
APPLICATION DATABASE
REPORT

FIG. 11

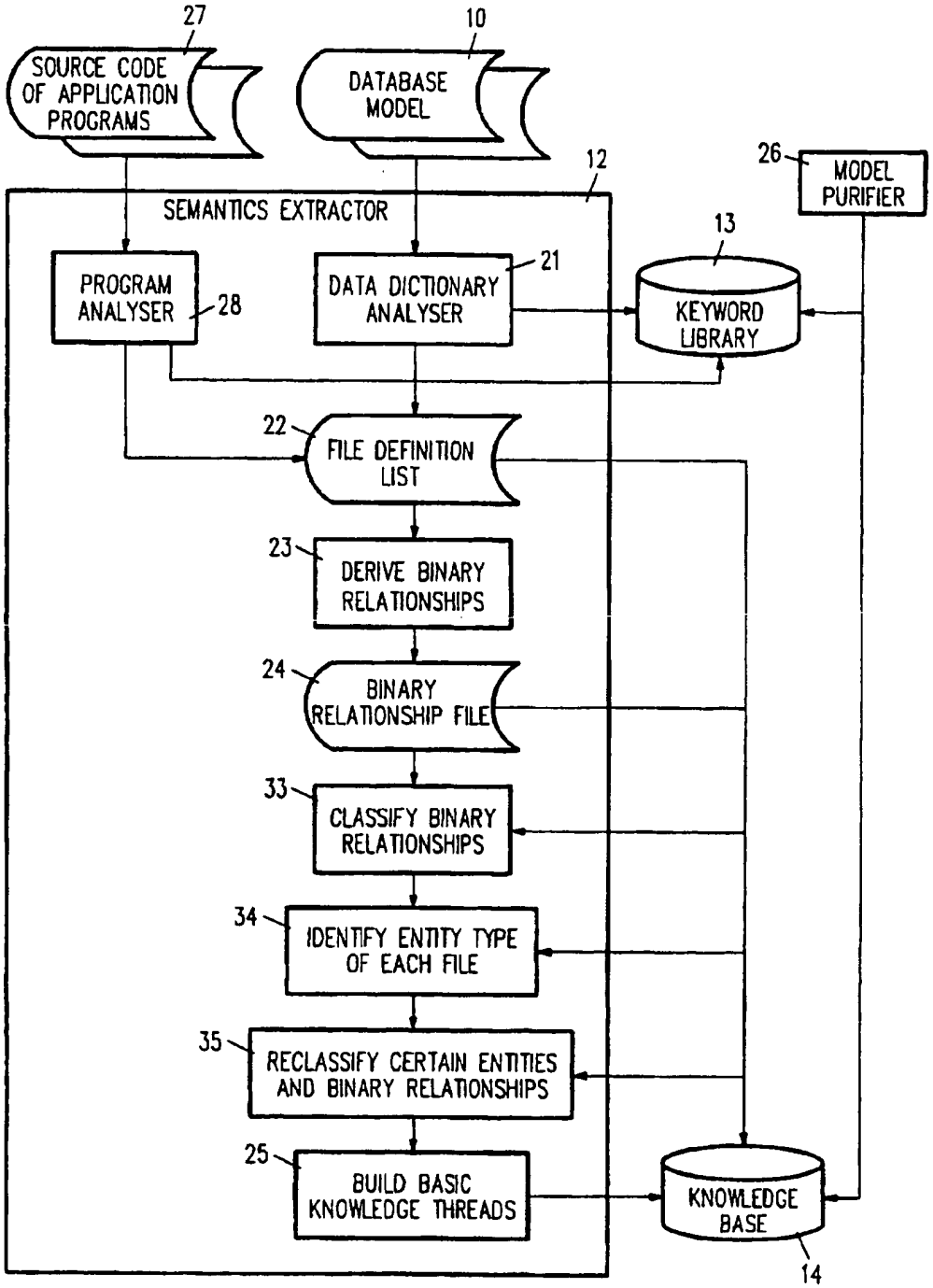

27
SOURCE CODE OF APPLICATION PROGRAMS
10
DATABASE MODEL
12
26
MODEL PURIFIER
SEMANTICS EXTRACTOR
13
PROGRAM ANALYSER
28
DATA DICTIONARY ANALYSER
21
KEYWORD LIBRARY
22
FILE DEFINITION LIST
23
DERIVE BINARY RELATIONSHIPS
24
BINARY RELATIONSHIP FILE
33
CLASSIFY BINARY RELATIONSHIPS
34
IDENTIFY ENTITY TYPE OF EACH FILE
35
RECLASSIFY CERTAIN ENTITIES AND BINARY RELATIONSHIPS
25
BUILD BASIC KNOWLEDGE THREADS
KNOWLEDGE BASE
14

FIG. 12

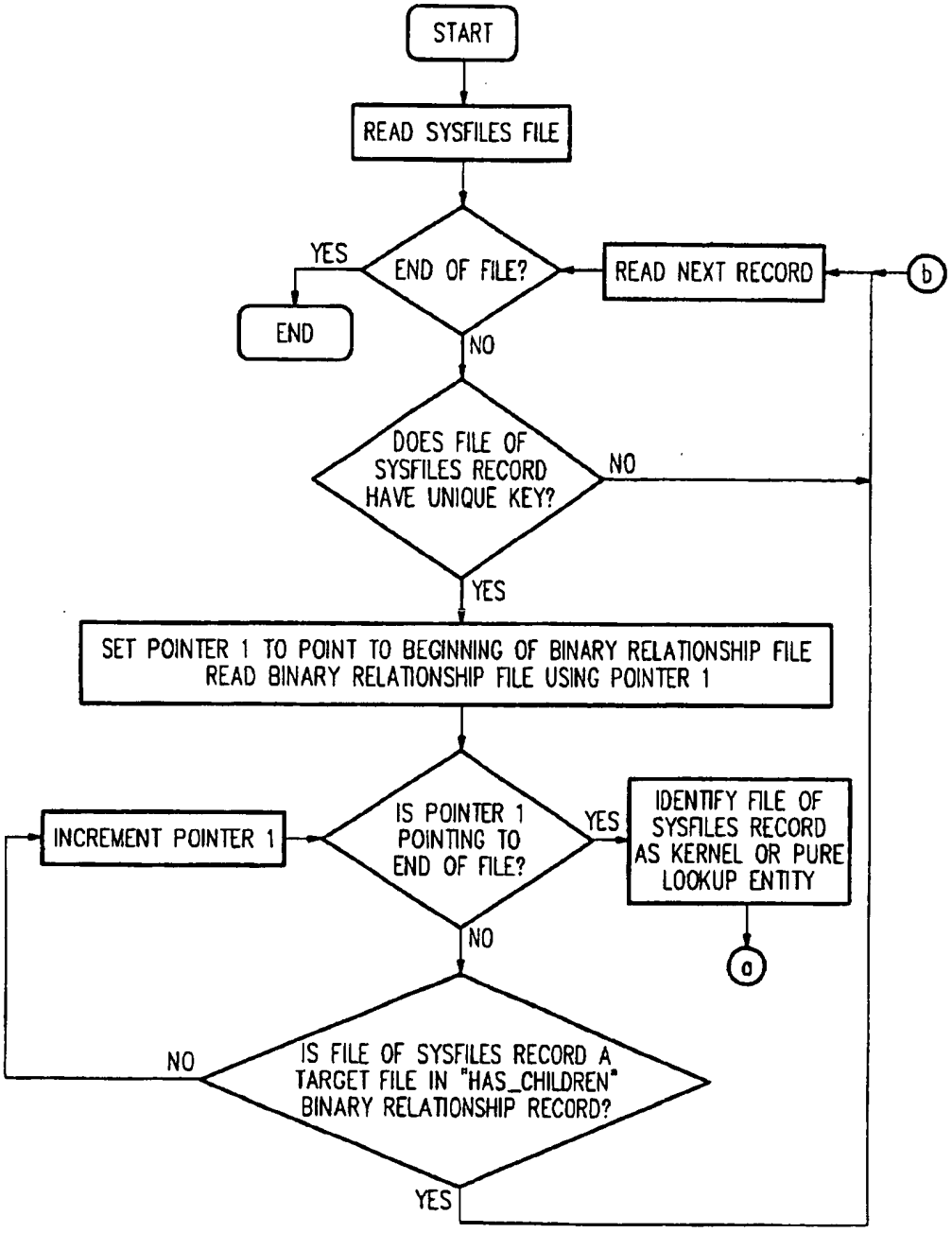
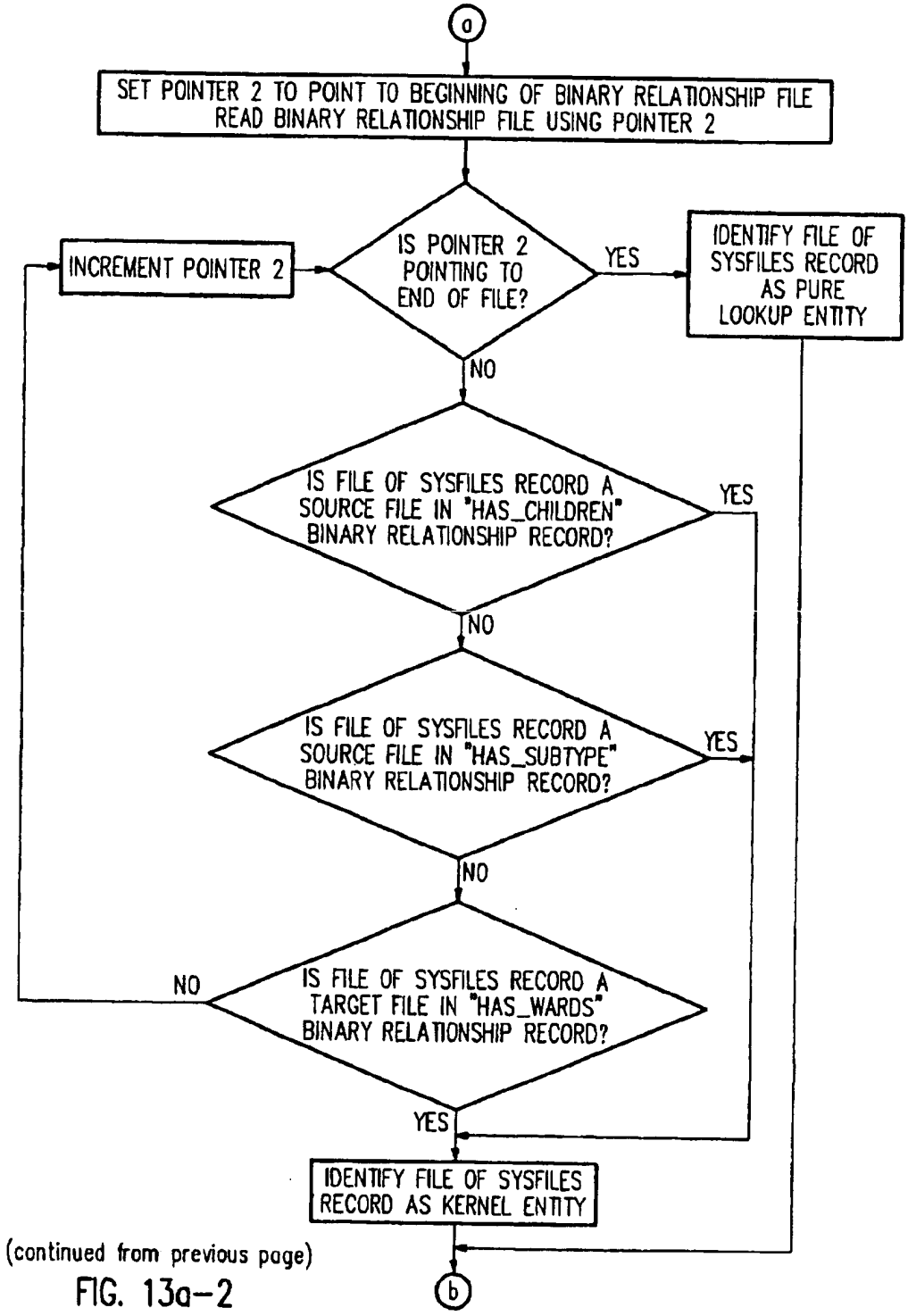

START
READ SYSFILES FILE
END OF FILE?
YES
END
READ NEXT RECORD
b
NO
DOES FILE OF SYSFILES RECORD HAVE UNIQUE KEY?
NO
YES
SET POINTER 1 TO POINT TO BEGINNING OF BINARY RELATIONSHIP FILE
READ BINARY RELATIONSHIP FILE USING POINTER 1
IS POINTER 1 POINTING TO END OF FILE?
YES
IDENTIFY FILE OF SYSFILES RECORD AS KERNEL OR PURE LOOKUP ENTITY
a
INCREMENT POINTER 1
NO
IS FILE OF SYSFILES RECORD A TARGET FILE IN "HAS_CHILDREN" BINARY RELATIONSHIP RECORD?
NO
YES

FIG. 13a-1



(continued from previous page)

FIG. 13a-2

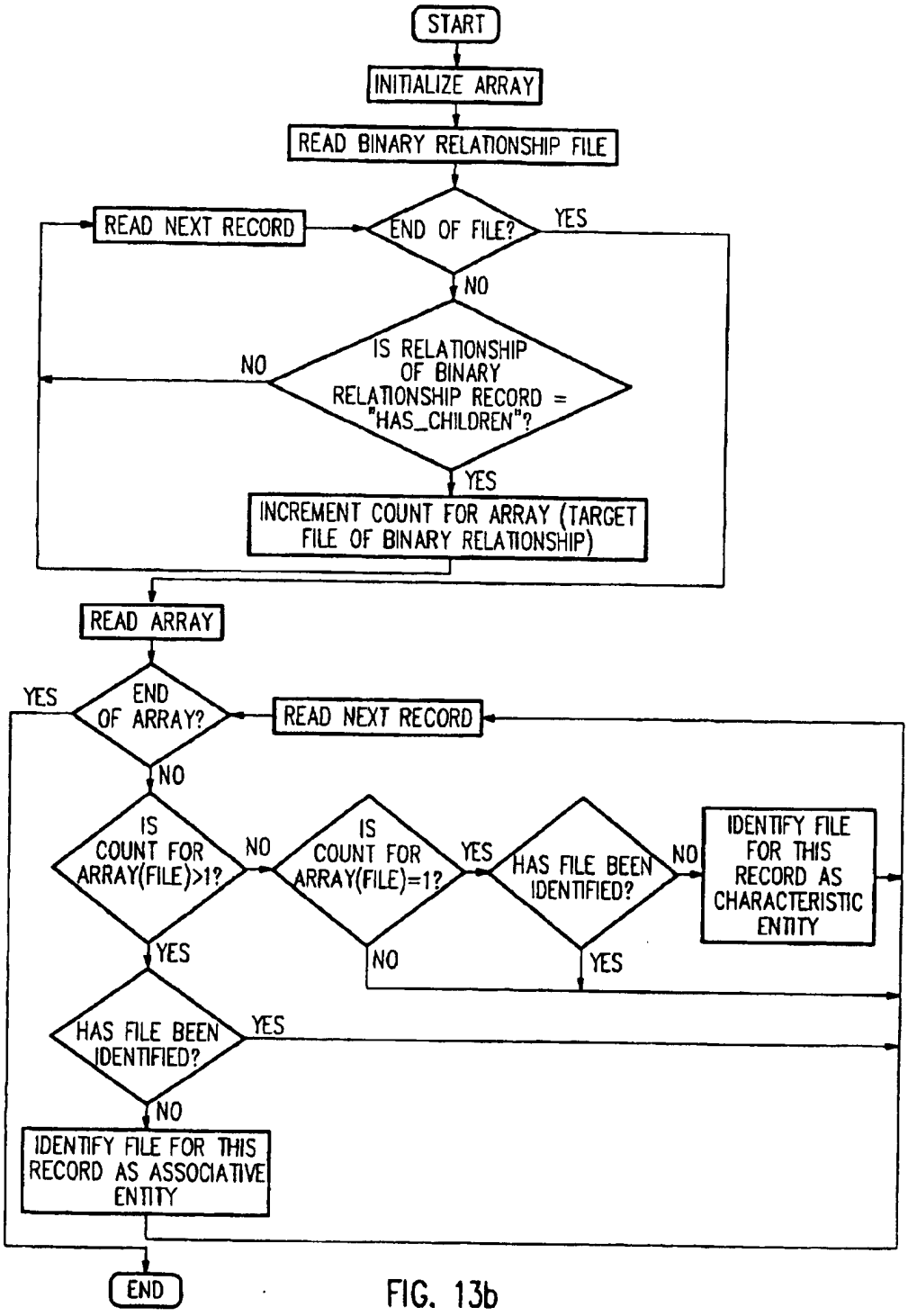

START
INITIALIZE ARRAY
READ BINARY RELATIONSHIP FILE
READ NEXT RECORD
END OF FILE?
YES
NO
IS RELATIONSHIP OF BINARY RELATIONSHIP RECORD = "HAS_CHILDREN"?
NO
YES
INCREMENT COUNT FOR ARRAY (TARGET FILE OF BINARY RELATIONSHIP)
READ ARRAY
YES
END OF ARRAY?
READ NEXT RECORD
NO
IS COUNT FOR ARRAY(FILE)>1?
NO
IS COUNT FOR ARRAY(FILE)=1?
YES
HAS FILE BEEN IDENTIFIED?
NO
IDENTIFY FILE FOR THIS RECORD AS CHARACTERISTIC ENTITY
YES
NO
YES
HAS FILE BEEN IDENTIFIED?
YES
NO
IDENTIFY FILE FOR THIS RECORD AS ASSOCIATIVE ENTITY
END

FIG. 13b

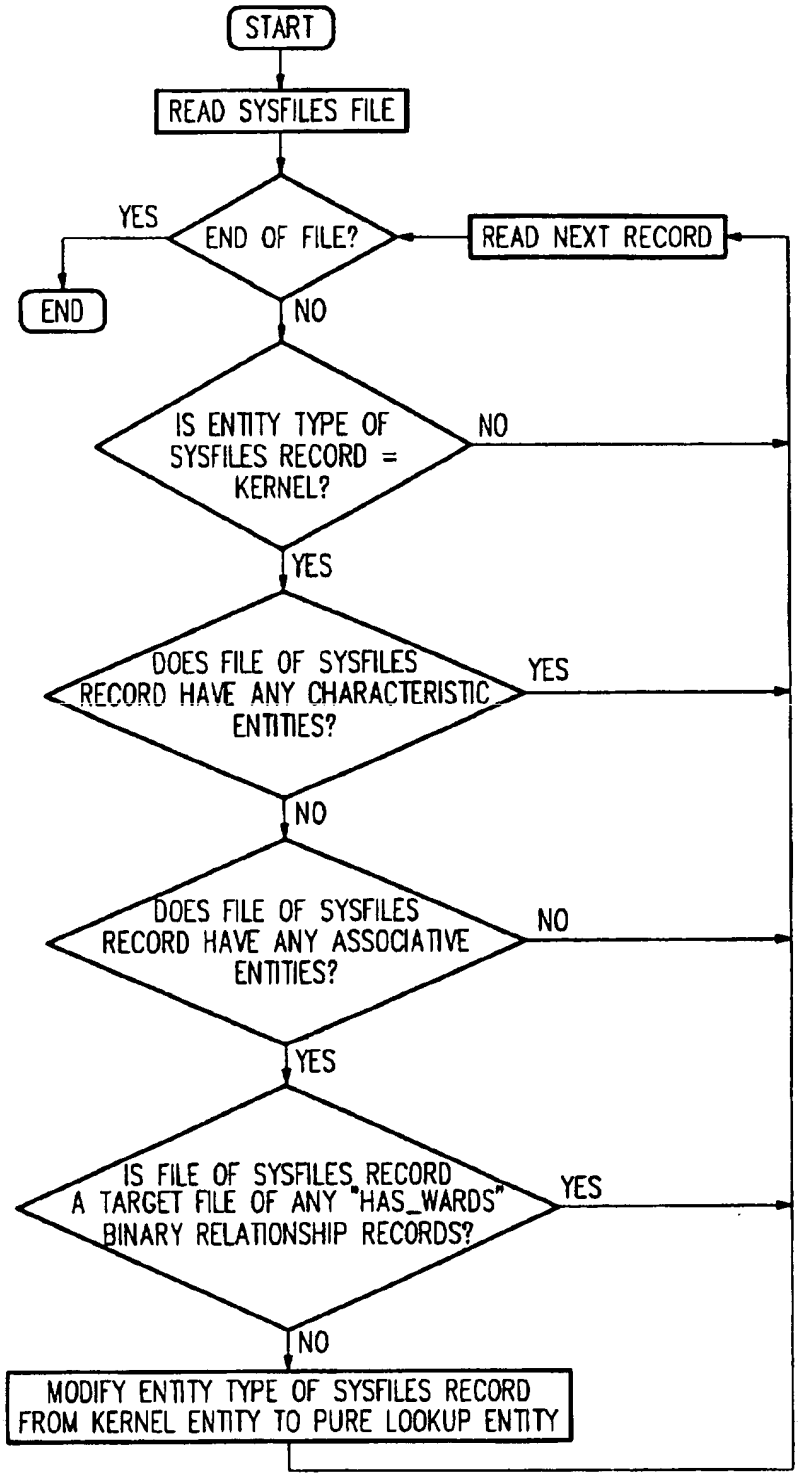

START
READ SYSFILES FILE
END OF FILE?
YES
END
NO
READ NEXT RECORD
IS ENTITY TYPE OF SYSFILES RECORD = KERNEL?
NO
YES
DOES FILE OF SYSFILES RECORD HAVE ANY CHARACTERISTIC ENTITIES?
YES
NO
DOES FILE OF SYSFILES RECORD HAVE ANY ASSOCIATIVE ENTITIES?
NO
YES
IS FILE OF SYSFILES RECORD A TARGET FILE OF ANY "HAS_WARDS" BINARY RELATIONSHIP RECORDS?
YES
NO
MODIFY ENTITY TYPE OF SYSFILES RECORD FROM KERNEL ENTITY TO PURE LOOKUP ENTITY

FIG. 14

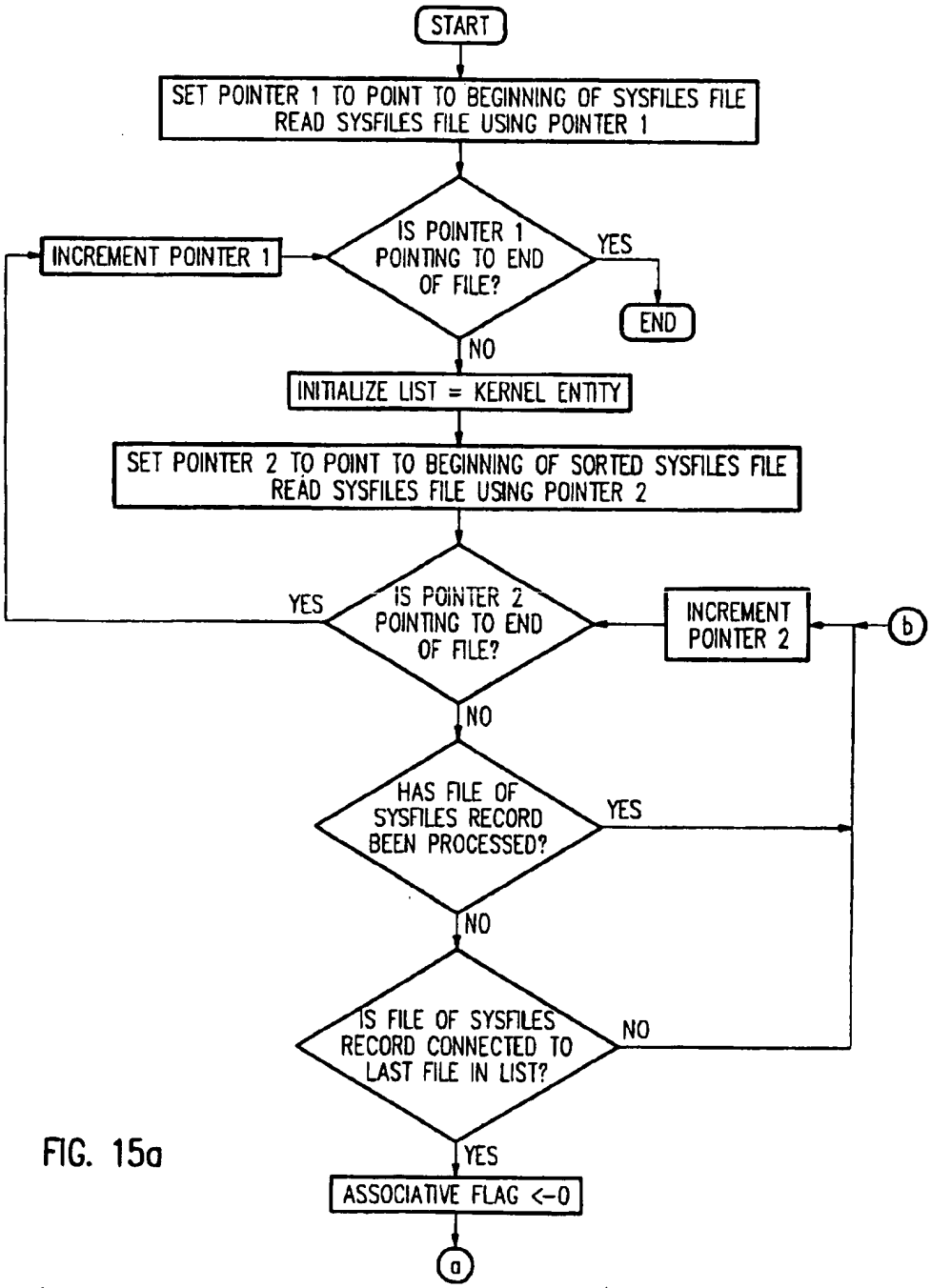

START
SET POINTER 1 TO POINT TO BEGINNING OF SYSFILES FILE
READ SYSFILES FILE USING POINTER 1
IS POINTER 1 POINTING TO END OF FILE?
YES
END
NO
INCREMENT POINTER 1
INITIALIZE LIST = KERNEL ENTITY
SET POINTER 2 TO POINT TO BEGINNING OF SORTED SYSFILES FILE
READ SYSFILES FILE USING POINTER 2
IS POINTER 2 POINTING TO END OF FILE?
YES
INCREMENT POINTER 2
b
NO
HAS FILE OF SYSFILES RECORD BEEN PROCESSED?
YES
NO
IS FILE OF SYSFILES RECORD CONNECTED TO LAST FILE IN LIST?
NO
YES
ASSOCIATIVE FLAG <-0
a

FIG. 15a

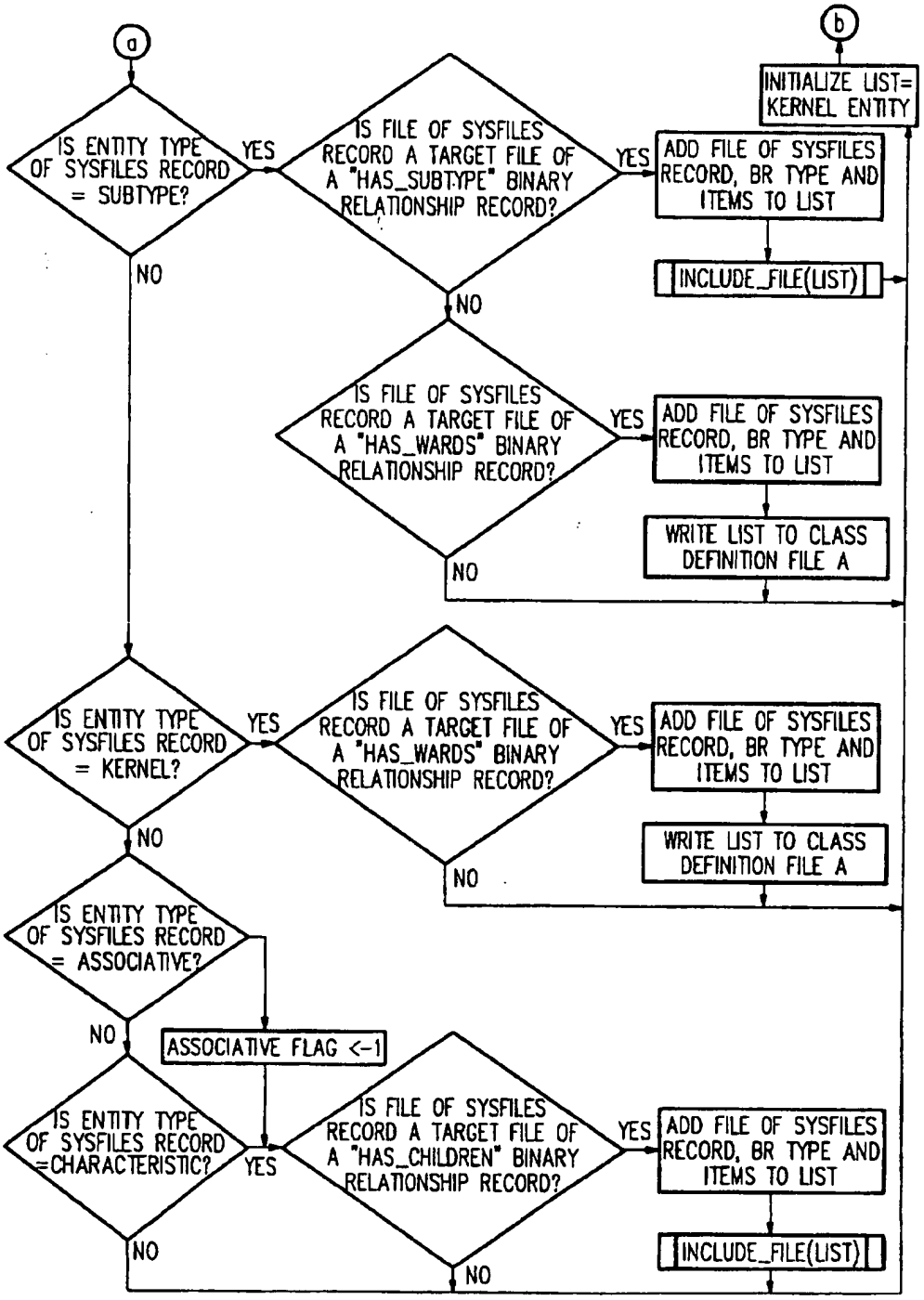


FIG. 15b (continued from previous page)

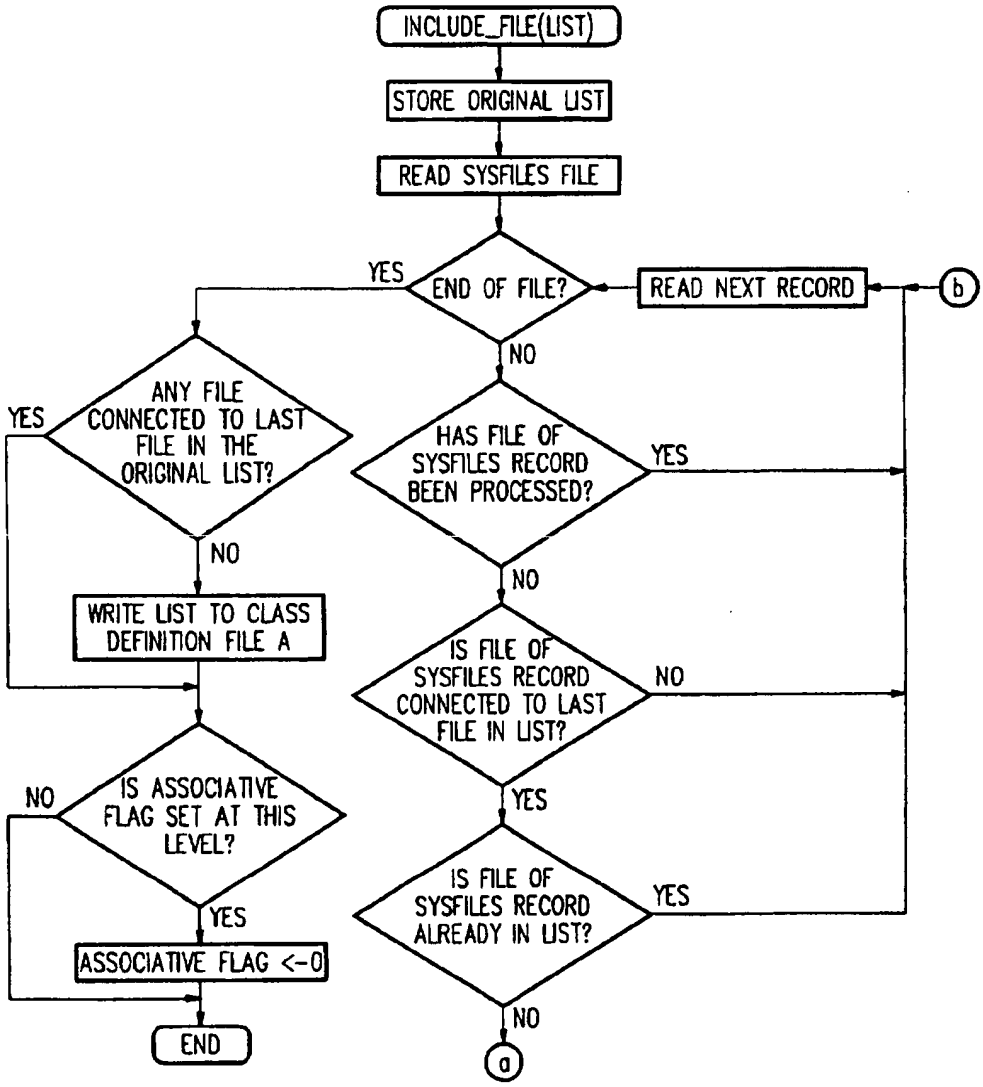
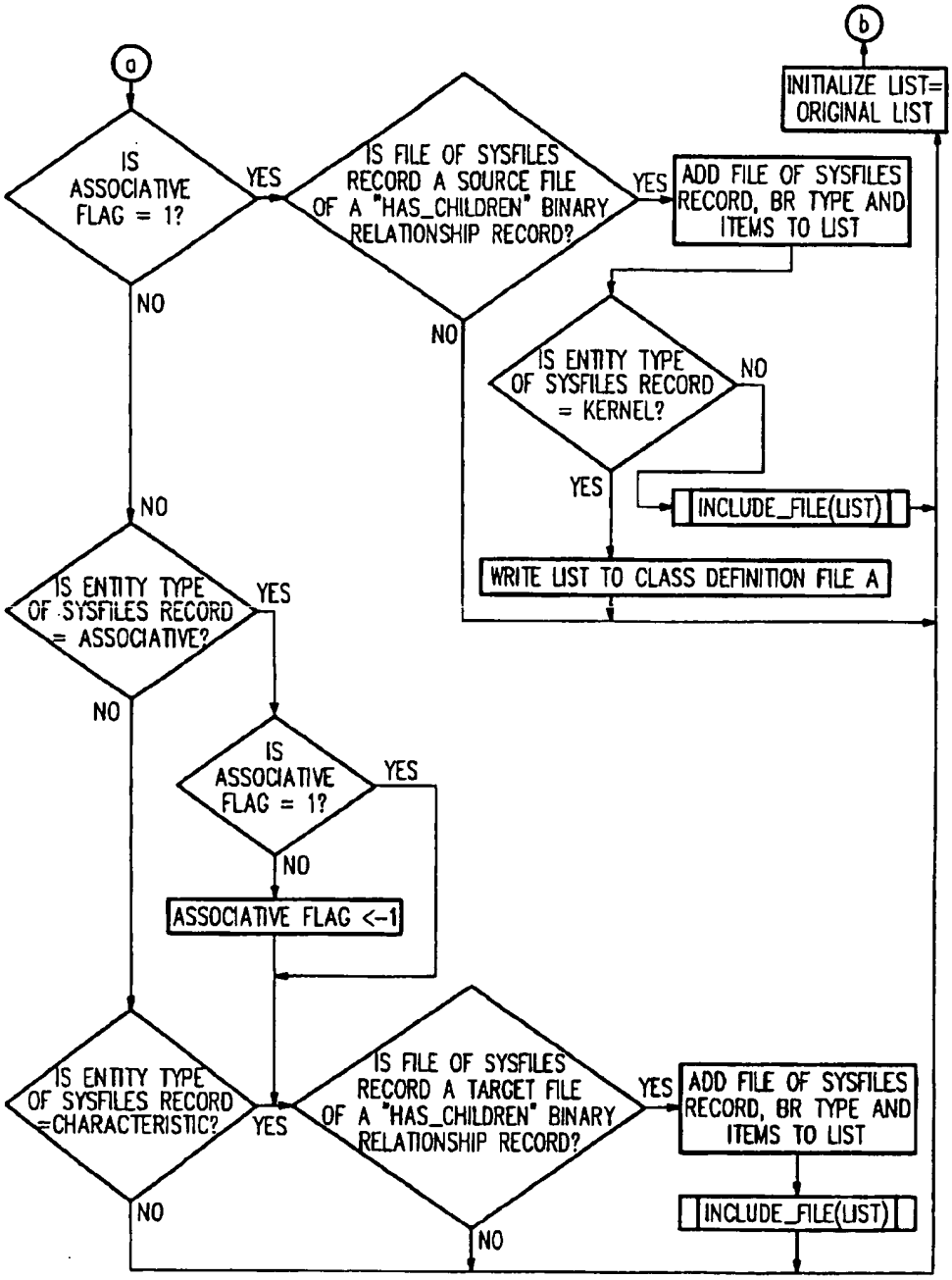

INCLUDE_FILE(LIST)
STORE ORIGINAL LIST
READ SYSFILES FILE
END OF FILE?
YES
READ NEXT RECORD
b
NO
ANY FILE CONNECTED TO LAST FILE IN THE ORIGINAL LIST?
YES
NO
WRITE LIST TO CLASS DEFINITION FILE A
IS ASSOCIATIVE FLAG SET AT THIS LEVEL?
NO
YES
ASSOCIATIVE FLAG <-0
END
HAS FILE OF SYSFILES RECORD BEEN PROCESSED?
YES
NO
IS FILE OF SYSFILES RECORD CONNECTED TO LAST FILE IN LIST?
NO
YES
IS FILE OF SYSFILES RECORD ALREADY IN LIST?
YES
NO
a

FIG. 16a



(continued from previous page)

FIG. 16b

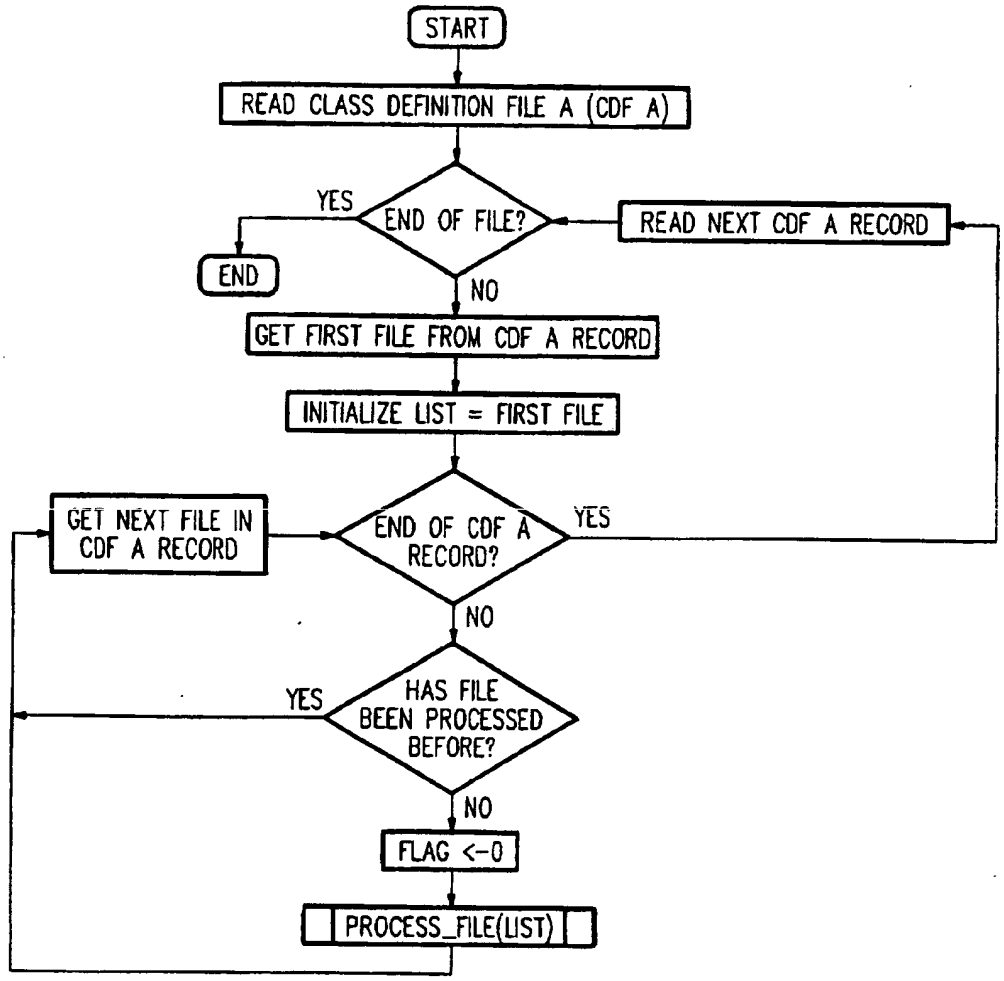

START
READ CLASS DEFINITION FILE A (CDF A)
END OF FILE?
YES
END
NO
READ NEXT CDF A RECORD
GET FIRST FILE FROM CDF A RECORD
INITIALIZE LIST = FIRST FILE
END OF CDF A RECORD?
YES
NO
GET NEXT FILE IN CDF A RECORD
HAS FILE BEEN PROCESSED BEFORE?
YES
NO
FLAG <-0
PROCESS_FILE(LIST)

FIG. 17

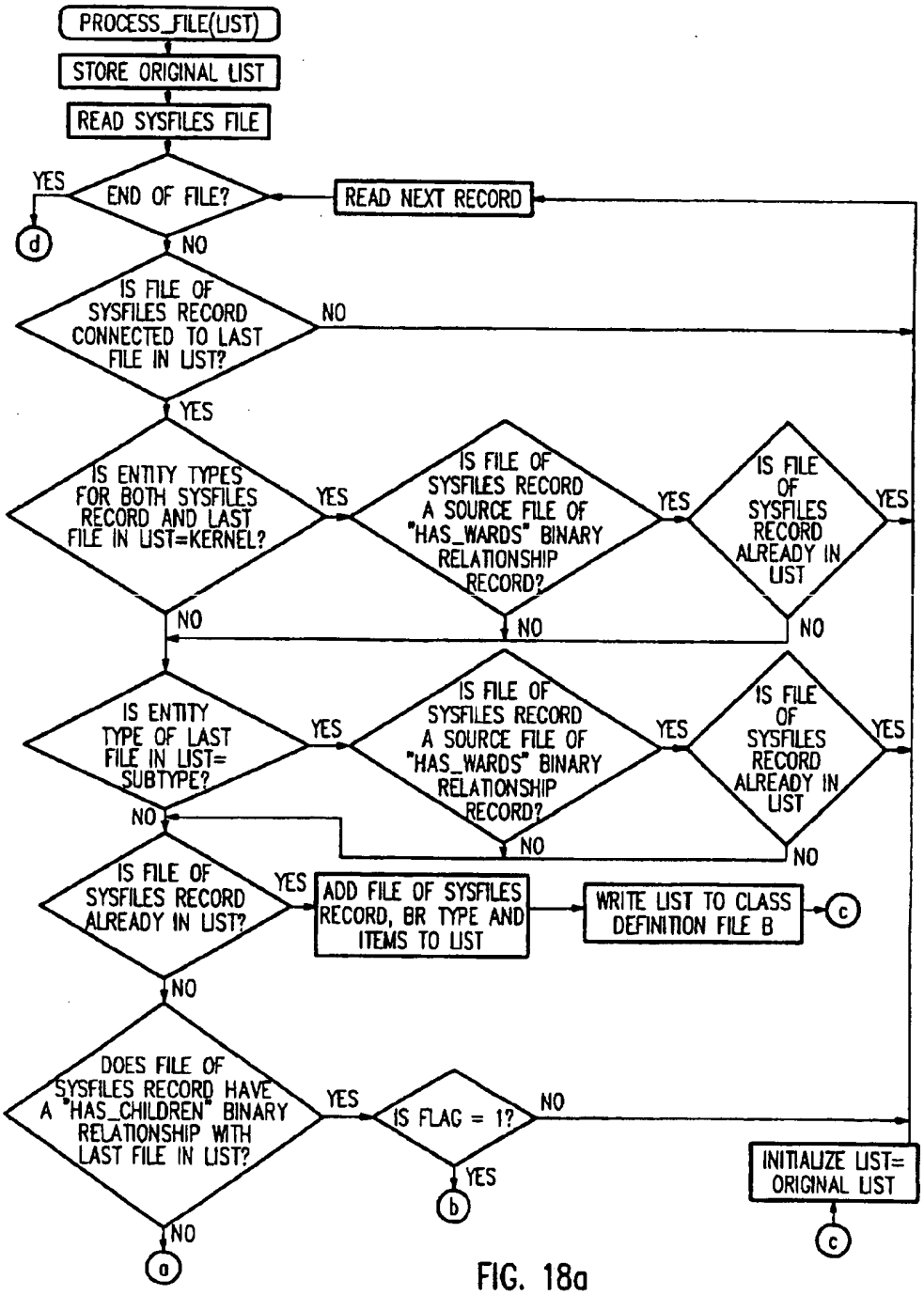
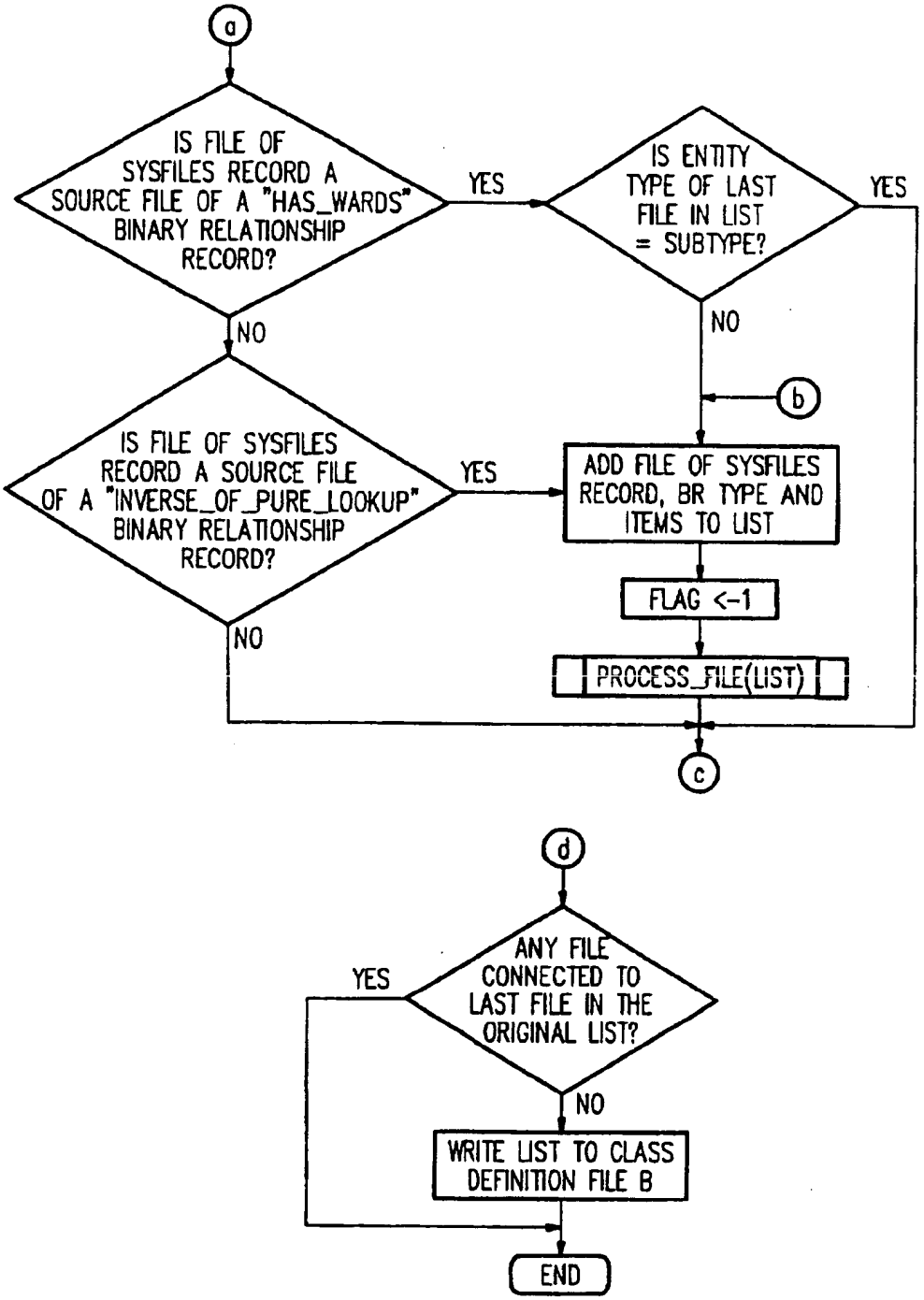

PROCESS_FILE(LIST)
STORE ORIGINAL LIST
READ SYSFILES FILE
END OF FILE?
YES
READ NEXT RECORD
NO
d
IS FILE OF SYSFILES RECORD CONNECTED TO LAST FILE IN LIST?
NO
YES
IS ENTITY TYPES FOR BOTH SYSFILES RECORD AND LAST FILE IN LIST=KERNEL?
YES
IS FILE OF SYSFILES RECORD A SOURCE FILE OF "HAS_WARDS" BINARY RELATIONSHIP RECORD?
YES
IS FILE OF SYSFILES RECORD ALREADY IN LIST
YES
NO
NO
NO
IS ENTITY TYPE OF LAST FILE IN LIST= SUBTYPE?
YES
IS FILE OF SYSFILES RECORD A SOURCE FILE OF "HAS_WARDS" BINARY RELATIONSHIP RECORD?
YES
IS FILE OF SYSFILES RECORD ALREADY IN LIST
YES
NO
NO
NO
IS FILE OF SYSFILES RECORD ALREADY IN LIST?
YES
ADD FILE OF SYSFILES RECORD, BR TYPE AND ITEMS TO LIST
WRITE LIST TO CLASS DEFINITION FILE B
c
NO
DOES FILE OF SYSFILES RECORD HAVE A "HAS_CHILDREN" BINARY RELATIONSHIP WITH LAST FILE IN LIST?
YES
IS FLAG = 1?
NO
YES
b
INITIALIZE LIST= ORIGINAL LIST
c
NO
a

FIG. 18a



(continued from previous page)

FIG. 18b

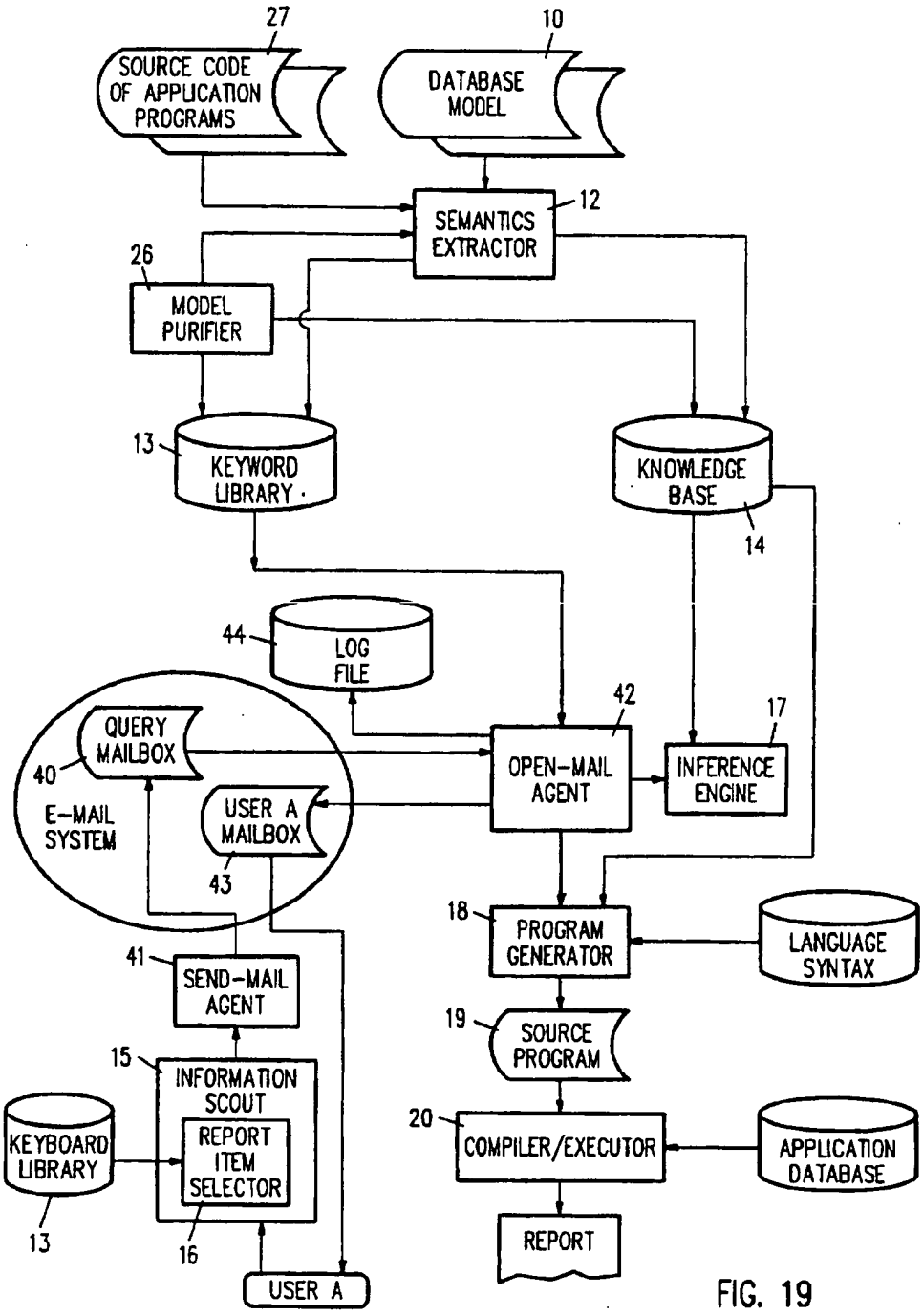

27
SOURCE CODE OF APPLICATION PROGRAMS
10
DATABASE MODEL
12
SEMANTICS EXTRACTOR
26
MODEL PURIFIER
13
KEYWORD LIBRARY
KNOWLEDGE BASE
14
44
LOG FILE
42
17
QUERY MAILBOX
40
E-MAIL SYSTEM
USER A MAILBOX
43
OPEN-MAIL AGENT
INFERENCE ENGINE
18
PROGRAM GENERATOR
LANGUAGE SYNTAX
41
SEND-MAIL AGENT
19
SOURCE PROGRAM
15
INFORMATION SCOUT
KEYBOARD LIBRARY
REPORT ITEM SELECTOR
20
COMPILER/EXECUTOR
APPLICATION DATABASE
13
16
REPORT
USER A

FIG. 19

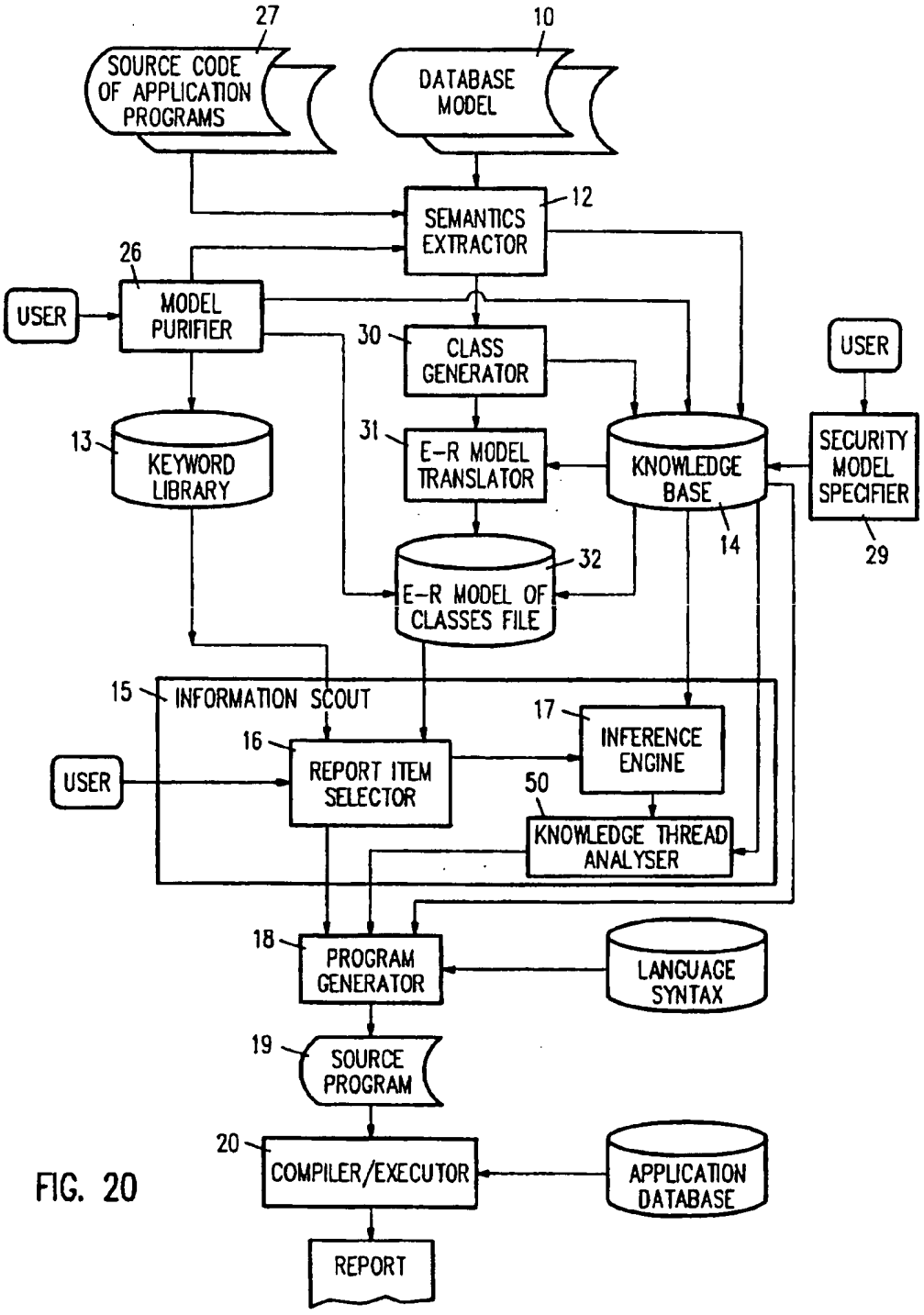

27
SOURCE CODE OF APPLICATION PROGRAMS
10
DATABASE MODEL
SEMANTICS EXTRACTOR
12
26
USER
MODEL PURIFIER
30
CLASS GENERATOR
USER
13
KEYWORD LIBRARY
31
E-R MODEL TRANSLATOR
KNOWLEDGE BASE
14
SECURITY MODEL SPECIFIER
29
32
E-R MODEL OF CLASSES FILE
15
INFORMATION SCOUT
17
INFERENCE ENGINE
16
USER
REPORT ITEM SELECTOR
50
KNOWLEDGE THREAD ANALYSER
18
PROGRAM GENERATOR
LANGUAGE SYNTAX
19
SOURCE PROGRAM
20
COMPILER/EXECUTOR
APPLICATION DATABASE
REPORT

FIG. 20

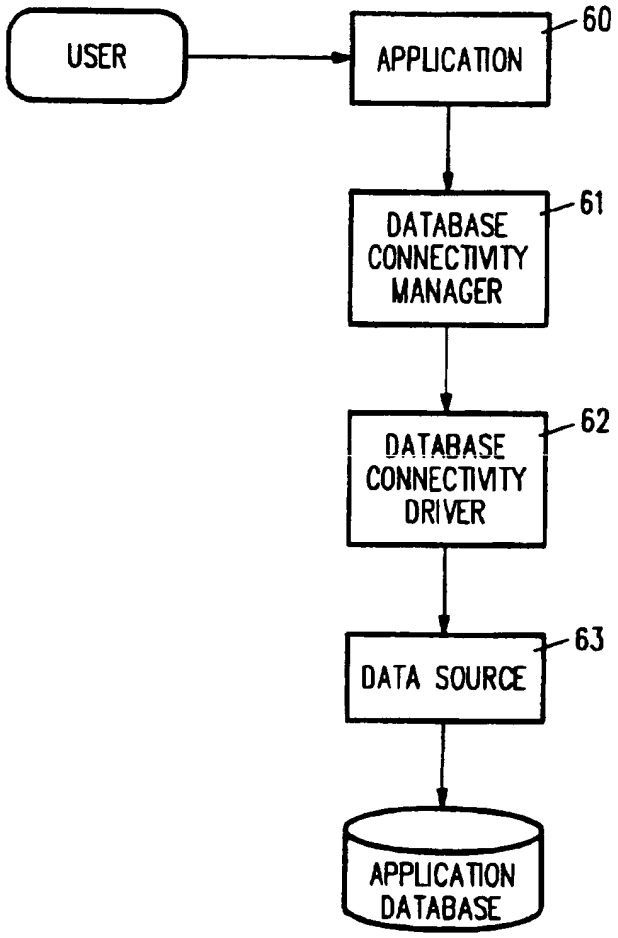

USER
APPLICATION
60
DATABASE CONNECTIVITY MANAGER
61
DATABASE CONNECTIVITY DRIVER
62
DATA SOURCE
63
APPLICATION DATABASE

FIG. 21

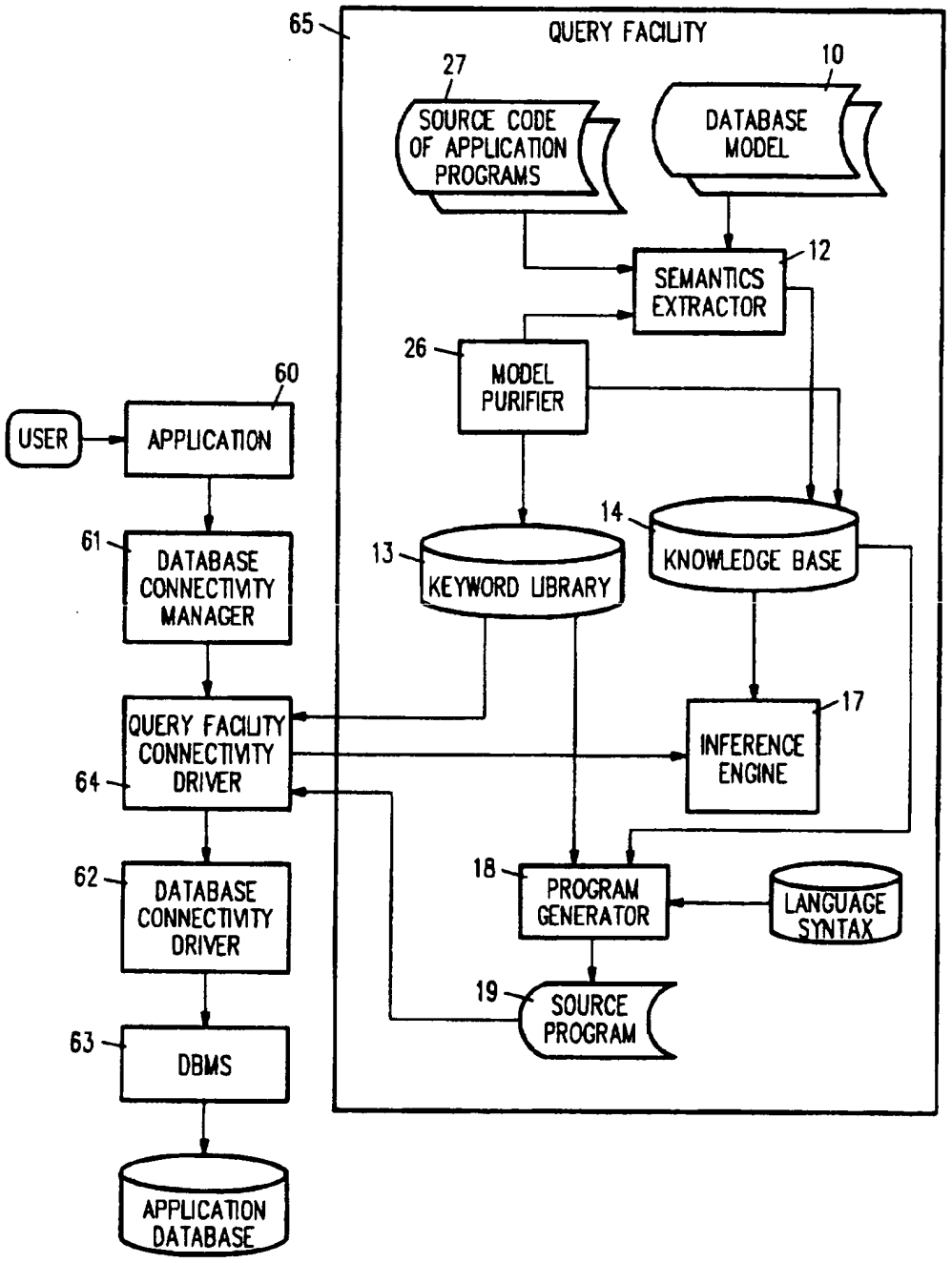

65
QUERY FACILITY
27
SOURCE CODE OF APPLICATION PROGRAMS
10
DATABASE MODEL
12
SEMANTICS EXTRACTOR
26
MODEL PURIFIER
60
USER
APPLICATION
61
DATABASE CONNECTIVITY MANAGER
13
KEYWORD LIBRARY
14
KNOWLEDGE BASE
QUERY FACILITY CONNECTIVITY DRIVER
64
17
INFERENCE ENGINE
62
DATABASE CONNECTIVITY DRIVER
18
PROGRAM GENERATOR
LANGUAGE SYNTAX
19
SOURCE PROGRAM
63
DBMS
APPLICATION DATABASE

FIG. 22

END USER QUERY FACILITY INCLUDING A QUERY CONNECTIVITY DRIVER

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation-in-part application of U.S. patent application Ser. No. 08/346,507, filed Nov. 29, 1994, (now U.S. Pat. No. 5,701,466), which is a continuation-in-part application of U.S. patent application Ser. No. 08/154,343 filed Nov. 17, 1993 (now abandoned in favor of continuation application Ser. No. 08/348,742, filed Nov. 30, 1994, now U.S. Pat. No. 5,487,132, issued Jan. 23, 1996), which in turn is a continuation-in-part of U.S. patent application Ser. No. 07/846,522, filed Mar. 4, 1992, (now U.S. Pat. No. 5,325,465, issued Jun. 28, 1994).

TECHNICAL FIELD

This invention pertains to end user query technology, and more specifically to an end user query facility which scouts for information by understanding the database model and guiding the user.

BACKGROUND

This invention pertains to end user query technology, introducing a novel approach to end user information access. Current end user query techniques require the user to understand database models in order to access information. For example, using prior art database models in which it is not uncommon to have dozens if not in excess of a hundred separate data base files interrelated, it is necessary for the database programmer to know in which file the desired piece of information is located, and then appropriately connect the files to achieve an orderly access of the specific file containing the desired information. This requires a fair amount of skill on the part of the database programmer and intimate familiarity of that programmer with the structure of the database which may be extremely complex. Furthermore, training new database programmers on an existing database model requires considerable amount of time and effort. One example of a prior art knowledge-based information retrieval system is the EASY-TALK product available from Intelligent Business Systems of Milford, Conn. However, it appears that the EASY-TALK product requires the database developer to explicitly input to the EASY-TALK system the semantics of the database.

SUMMARY OF THE INVENTION

In accordance with the teachings of this invention, it has been determined that there would be great usefulness in providing an end user query technology which is capable of automatically understanding the database model and guiding the user to scout for the desired information, thereby increasing productivity and ease of information access. In accordance with the teachings of this invention, the user is freed from the need to understanding the database model, with the end user query facility of this invention quickly guiding the user to acquire the information. This is made possible by the end user query facility of this invention first recapturing the application semantics from the existing database model to provide a set of derived semantics. The derived semantics are then used by the end user query facility to intelligently guide the user to scout for the desired information in the database. In addition, the derived semantics can be easily updated by the end user query facility when the database model is changed.

In accordance with further teachings of this invention, the user is provided with a "natural" description of the database model to further ease his effort in information access. The "natural" description includes classes with an entity-relationship (E-R) model describing each class. The classes represent the different types of high-level objects whose information are contained in the database. The description of these classes is made possible by the end-user query facility first recapturing additional application semantics from the existing database model to provide a richer set of derived semantics. This enriched set of derived semantics is then used to identify the classes and generate their definitions. The definition of each class is then subsequently translated into an E-R model of the class. In addition these classes and their E-R models can be easily updated by the end-user query facility when the database model is changed.

In accordance with the teachings of this invention, it has also been determined that there would be some usefulness in providing an end-user query technology that allows a user at a remote site with no on-line access to the database to still be able to make a query. This is made possible by integrating the end-user query facility of this invention with an electronic mail system so that the user at the remote site can send his query as a mail message and have the result of his query posted to him also as a mail message. In addition, a log of all query requests and their processing can be kept and analyzed to track usage and performance of the end-user query facility.

A key benefit of an additional embodiment of the present invention allows the many millions of existing applications used by many organizations such as Microsoft Excel spreadsheet, Microsoft Access DBMS, Lotus 123 spreadsheet, Powersoft PowerViewer or InfoMaker, Gupta Quest, Q+E Database Editor, etc. that are ODBC compliant or compliant to other data access interface standard to be reused for making powerful ad-hoc queries easily through seamless integration of a novel Query Facility provided by this invention, thus saving on the cost of purchasing new tools to have more powerful ad-hoc queries as well as the cost of training users to use the new tools.

BRIEF DESCRIPTION OF THE DRAWING

FIG. **1** is a flow chart depicting one embodiment of an end user query facility constructed in the accordance with the teachings of this invention;

FIG. **2** is a flow chart depicting one embodiment of semantics extractor **12** of FIG. **1**;

FIG. **3** is a flow chart depicting one embodiment of a method suitable for use to build basic knowledge threads step **25** of semantics extractor **12** of FIG. **2**;

FIG. **4** is a flow chart depicting one embodiment of a method suitable for use as information scout **15** of the embodiment to FIG. **1**;

FIG. **5** is a flow chart depicting one embodiment of a method suitable for use as the search knowledge base step of the embodiment to FIG. **4**;

FIGS. **6***a* and **6***b* is a flow chart depicting one method suitable for use as the infer new threads from knowledge base step of the embodiment of FIG. **4**;

FIG. **7** is a flow chart depicting an alternative embodiment of this invention including a model purifier in accordance with this invention;

FIG. **8** is a flow chart depicting one embodiment of a semantics extractor constructed in accordance with the teachings of this invention interfacing with a model purifier;

FIG. 9 is a flow chart depicting an alternative embodiment of a semantics extractor including a program analyzer in accordance with the teachings of this invention;

FIG. 10 is a flow chart depicting an alternative embodiment of this invention including a security model specifier;

FIG. 11 is a flow chart depicting an alternative embodiment of this invention which includes a model purifier, a security model specifier, an alternative embodiment of a semantics extractor, a class generator, and an entity-relationship (E-R) model translator;

FIG. 12 is a flow chart depicting an alternative embodiment of a semantics extractor to extract additional semantics in the form of types of binary relationships and entity types of files in order for the class generator to generate class definitions;

FIGS. 13*a*-1 and 13*a*-2 form a flow chart depicting one embodiment of a procedure to identify kernel entities, suitable for use in the step of identify entity type of each file in the embodiment of FIG. 12;

FIG. 13*b* is a flow chart depicting one embodiment of a procedure to identify characteristic and associative entities, suitable for use in the step of identify entity type of each file in the embodiment of FIG. 12;

FIG. 14 is a flow chart depicting one embodiment of a procedure to reclassify kernel entities into pure lookup entities, suitable for use in the step of reclassify certain entities and binary relationships in the embodiment of FIG. 12;

FIGS. 15*a* and 15*b* depict a flow chart of one embodiment of a procedure to generate class definition file A, suitable for use by the class generator in the embodiment of FIG. 11;

FIG. 17 is a flow chart depicting one embodiment of a procedure to generate class definition file B, suitable for use by the class generator in the embodiment of FIG. 11;

FIGS. 18*a* and 18*b* depict a flow chart of one embodiment of a procedure for Process_File suitable for use with the embodiment of FIG. 17; and

FIG. 19 is a flow chart depicting an alternative embodiment of this invention in which a query facility is integrated with an E-mail system;

FIG. 20 is a flowchart depicting one embodiment of this invention which includes a Knowledge Thread Analyzer;

FIG. 21 is a diagram depicting the relationship between the four components of the ODBC architecture; and

FIG. 22 is a diagram showing a Query Facility linked to an application using a Query Facility Connectivity Driver, in accordance with one embodiment of this invention.

DETAILED DESCRIPTION

Overview

The following briefly describes one embodiment of this invention which is in the reporting language known as QUIZ. The language is part of a fourth generation language known as POWERHOUSE from COGNOS Incorporated of Canada. However, it is to be understood that the teachings of this invention are equally applicable to any end user query technology, including languages other than QUIZ.

FIG. 1 depicts the operation of and interaction between the following modules:

a. The Semantics Extractor. Semantics Extractor **12** reads Database Model **10** of an application system and extracts the semantics of the database model. The semantics are stored in Keyword Library **13** as a list of keywords based on the item definitions and in Knowledge Base **14** as a list of file linkages of the database model. This is a key feature of the present invention and differs from the prior art in that prior art systems require a database developer or manager to explicitly define and provide the semantics of the database to the end user query facility.

b. The Information Scout. Information Scout **15** guides the user to specify the items to be reported in order to obtain the information he wants. This is carried out in two steps. First, Report Item Selector **16** prompts the user for keywords that suggest the desired data item, for example DATE. Using a look-alike keyword search method, all items in Keyword Library **13** containing DATE are listed. The user then makes the appropriate selection. After selecting the items, Inference Engine **17** identifies the files containing the selected items. Inference Engine **17** then searches for the linkage(s) in Knowledge Base **14** connecting the identified files.

c. The Program Generator. Program Generator **18** accesses the linkages obtained by Information Scout **15** and generates the corresponding Source Program **19** to extract the information requested by the User.

d. The Compiler/Executor. The source program is compiled and executed against the database to generate the report using Compiler/Executor **20**.

A more detailed description of one embodiment of these modules is now provided.

The Semantics Extractor

FIG. 2 shows a flow chart of Semantics Extractor **12**. In order to extract the semantics of an application, an application system must have a data dictionary that represents the application data model. In the POWERHOUSE fourth generation language which is used in one embodiment, all data dictionaries are available and can be found in "QDD" or "PDL" formats. They are created by the analyst when developing an application system. The data dictionary is made up of system options, security and definitions of each element, file, record and item used in an application system. The data dictionary of Database Model **10** is parsed by Data Dictionary Analyzer **21** in order to obtain the keywords and the information about each file and item in Database Model **10**.

The keywords are derived from the item definitions in the data dictionary in the following manner. The name of each item may be in the form of one complete word (e.g. SALARY) or may use more than one word separated with a hyphen (e.g. DATE-JOINED). Hyphens are removed from those item names with hyphens by Data Dictionary Analyzer **21**. The resulting individual words obtained from the item names are then stored as keywords in Keyword Library **13**, e.g. SALARY, DATE, and JOINED. These keywords are used later by Report Item Selector **16** (FIG. **1**).

Next, Knowledge Base **14** is built. The first step involves extracting the following information from each file in Database Model **10**:

a. File name

b. For every item in each file
   i) Item name
   ii) Item type (e.g. character, numeric, date)
   iii) Item size
   iv) If an item is a key, its key type (e.g. unique key, repeating key) or non-key

These files make up File Definition List **22** which is then used to determine the file relationships using the step Derive Binary Relationships **23**. This step involves comparing each file definition in File Definition List **22** with the other file definitions in the same list to obtain its binary relationships

with other files. A binary relationship consists of two files that can be linked. Two files can be linked if:

(i) both files have one item with the same name, AND

(ii) the item in the target (second) file is a unique or repeating key.

For example, in a Personnel Information System, the EMPLOYEES file would have an item called Emp-no which is a unique key to identify each employee and the SKILLS file would also have an item called Emp-no. Each record in a SKILLS file would contain a particular skill an employee has. As a single employee would have many different skills the Emp-no would be repeated for each skill and so would be a repeating key. These two files would form a binary relationship since Emp-no is the common item and Emp-no in SKILLS file is a repeating key. The item Emp-no in the source (first) file which in this example is the EMPLOYEES file need not be a key.

Often two items could be coded differently but mean the same thing, for example, both P-NO and PART-NO could be used to represent a part number in a Inventory Control System. P-NO may occur in some files and PART-NO in others. These two items are said to be in the same "domain" called P-NO as shown below:

File name	Item name	Domain
A-PARTS-FILE	P-NO	P-NO
B-PARTS-FILE	PART-NO	P-NO

In such a case, we would not be able to establish a file relationship between the two files as the item names are different. But it would still be meaningful to establish a link between the two files with such items as the items are in the same domain.

In the example above, both A-PARTS-FILE and B-PARTS-FILE have items in the same domain called P-NO though the item names are different. A-PARTS-FILE and B-PARTS-FILE should therefore be linked. The linkage rule defined earlier is thus modified as follows to take into account items with not only the same name but with different names in the same domain:

(i) both files have one item in the same domain, AND

(ii) the item in the target (second) file is a unique or repeating key.

For every linkage between two files, the item in the source file can be a unique key, repeating key or non-key while the target file must either be a unique or repeating key. From this restriction we can derive six possible valid types of file linkages. These are as follows:

	Source file	Target file	Notation
a.	Unique key	Unique key	UU
b.	Unique key	Repeating key	UR
c.	Repeating key	Unique key	RU
d.	Repeating key	Repeating key	RR
e.	Non-Key	Unique key	NU
f.	Non-Key	Repeating key	NR

However, in one embodiment the repeating to repeating (RR) combination which is item d above is not stored because this represents a bad file design. A repeating to repeating relationship indicates a many to many relationship which preferably should not exist in a normalized data model. The analyst is informed of such a finding and attempt to rectify it.

All the binary relationships found using the above rules are stored as follows:

a. Source file

b. Source file item to link to target file

c. Target file

d. Target file item to link to source file

e. Relationship (e.g. UU,UR,RU,NU,NR)

From the earlier example using EMPLOYEES and SKILLS files, the binary relationship are stored as:

Source file	Item	Target file	Item	Relationship
EMPLOYEES	Emp-no	SKILLS	Emp-no	UR

The set of binary relationships derived from the above step is stored in a Binary Relationship File **24**. The next step, namely Build Basic Knowledge Threads **25**, involves deriving knowledge of Database Model **10** of an application system from these binary relationships which is then stored in Knowledge Base **14** in the form of knowledge threads. Each thread represents a linkage of many files. The following is an example of a knowledge thread:

EMPLOYEES→BRANCHES→EXPENSES

It contains an EMPLOYEES file linked to BRANCHES file which is then linked to EXPENSES file. Inference Engine **17** (FIG. **1**) in Information Scout **15** later uses these threads in isolation or combination to infer the access paths in order to obtain the information requested by the user. For example, when the user wants to report the employees and their expenses in each branch, the above thread is used to generate the required path to navigate through an application database in order to acquire the required information.

Knowledge Base **14** is made up of basic and acquired knowledge threads. The following describes the derivation of basic knowledge threads by joining binary relationships from Binary Relationship File **24**. The derivation of acquired knowledge threads is described later.

FIG. **3** is a flow chart of one method suitable for use as Build Basic Knowledge Threads **25** of FIG. **2**. The following describes how a single basic knowledge thread is built using the method of FIG. **3**, with reference to the following example of a procedure Find-Thread:

```
Procedure Find-Thread (file-count, pointer 2)
begin
  old-file-count = file-count
  while (pointer 2 <> end of file)
  begin
    read binary relationship file using pointer 2
    until (thread-end = source file of binary relation
record)
        or (end of binary relationship file)
  if (binary relation record exists) and
    (target-file of record not exist in current-thread)
  then begin
        call Procedure Find-Thread(file-count, pointer 2)
        file-count = file-count + 1
        if (max-count < file-count)
        then max-count = file-count
        current-thread = current-thread + target-file
        reset pointer 2 to start of binary relationship
    file endif
  endwhile
  if file-count <> old-file-count
  then write current-thread into Basic Knowledge file
end
```

Note: Parameters file-count and pointer 2 are passed by value Current-thread, max-count are global variables

The formation of each thread begins with a first binary relationship record read from Binary Relationship File **24**. This record forms an initial thread. However, if the first

binary relationship record is of type 'RU' (repeating to unique) relationship. it is ignored and no initial thread is formed from it. The next record is then read and if it still is of type 'RU'. it is again ignored until the next record is found that is not of type 'RU'. which is then used to form the initial thread. The reason for doing this is that it is only necessary to form threads using either 'RU' or 'UR' types and not both as they lead to the same access path being inferred by Inference Engine **17**. In this embodiment we are using 'UR' types. To keep track of the next record to read from Binary Relationship File **24**. a pointer called pointer 1 is used.

The initial thread can be extended by linking with other valid binary relationship records. To do so, Binary Relationship File is searched. This time another pointer, namely pointer **2**. is used to keep track of the next record to read from Binary Relationship File **24** to link to the thread to be extended. A link to extend the thread is said to be found if the first file of the binary relationship record and the file at the end of the thread. called the thread-end file. are the same. or else the next binary relationship record is read using pointer **2**. If a valid binary relationship record is found. we then examine if the target (second) file of this record is already in the thread. If it is not. it is added to the end of the thread. to become the thread-end. If it is. the next binary relationship record is read using pointer **2**. The search ends when the end of Binary Relationship File **24** is reached using pointer **2**. The formation of a single thread is then complete and is stored in Knowledge Base **14** as follows:

Thread-Head:
  file name
  file item to link to next file

Thread-Body:
  file name
    file item to link to the previous file
    relationship to the previous file
  file item to link to the next file

Thread-End:
  file name
    file item to link to the previous file
  relationship to the previous file

The thread-head contains the first file and the first file item. It is linked to the next file using the thread-body or thread-end. The thread-body is made up of a file. its item which has the same domain as the item of the previous file it is linked to and the relationship to that file. It also contains another item which is used to link to the next file. The thread-end is similar to the thread-body except that it does not have the item to link to the next file.

The following example briefly shows how Knowledge Base **14** is built up with basic knowledge threads using the above procedure. Assume that Binary Relationship File **24** contains the following records:

Source file	Item	Targret file	Item	Relationship
1. EMPLOYEES	Br-no	BRANCHES	Br-no	NU
2. BRANCHES	Br-no	EXPENSES	Br-no	UR
3. EMPLOYEES	Emp-no	BILLINGS	Emp-no	UR
4. BILLINGS	Emp-no	EMPLOYEES	Emp-no	RU
5. PROJECTS	Proj-no	BILLINGS	Proj-no	UR
6. BILLINGS	Proj-no	PROJECTS	Proj-no	RU

The method of this example is then as follows:

Pointer 1 at 1st record of above file	Thread Formed Use the first binary relationship record EMPLOYEES link to BRANCHES as the valid binary relationship record to form the initial thread. Next, examine whether this thread can be extended by searching through the same Binary Relationship File, this time using pointer 2. It can be linked to the second binary relationship record, namely BRANCHES link to EXPENSES to form the final thread: EMPLOYEES → BRANCHES → EXPENSES
2nd record	Similarly, using the next binary relationship BRANCHES link to EXPENSES record, form the initial thread: BRANCHES → EXPENSES This thread cannot be extended as the EXPENSES file cannot be linked to other binary relationship records.
3rd record Using the 3rd binary relationship the third thread EMPLOYEES→ BILLINGS→ PROJECTS is formed as follows: EMPLOYEES links to BILLINGS via Emp-no BILLINGS links to PROJECTS via Proj-no This thread uses different items, namely Emp-no and Proj-no to link the two binary relationship records.	
4th record	No thread is formed as relationship is 'RU'.
5th record	Thread formed is PROJECTS → BILLINGS → EMPLOYEES as follows: PROJECTS links to BILLINGS via Proj-no BILLINGS links to EMPLOYEES via Emp-no This thread uses different items, namely Proj-no and Emp-no, to link the two binary relationship records.
6th record	No thread is formed as relationship is 'RU'.

Each thread is stored in Knowledge Base 14 as follows, using the third thread above as example:

Thread-Head file = EMPLOYEES
item = Emp-no
Thread-Body file = BILLINGS
item-1 = Emp-no
relationship = UR
item-2 = Proj-no
Thread-End file = PROJECTS
item = Proj-no
relationship = RU

Information Scout

FIG. **4** is a flow chart of one method suitable for use as Information Scout **15** of FIG. **1**. Report Item Selector **16** first guides the user to select the items to be reported. Inference Engine **17** then infers the access path based on the items selected.

Report Item Selector

The user is prompted for a keyword or presented with a list of all the words in Keyword Library **13** to choose from. When the user chooses to provide a keyword, Report Item

Selector 16 lists all the items that match the keyword provided. For example, the keyword DATE could return the following list of items containing the word DATE.

DATE BIRTH
DATE JOINED
DELIVERY DATE
LAST DATE UPDATE

The user then selects the desired item from the list. In one embodiment, an explanation of each item is also displayed. This explanation is either extracted from the data dictionary of an application system by Data Dictionary Analyzer 21 or entered by a programmer analyst maintaining the application system.

Using the above method, the user selects all the items to be reported. For each item, the file containing it is automatically identified and added to the file list using Generate File List Step 27. When two or more items are selected from the same file, only one entry is made in the file list. This file list is sorted such that the first file is the file which has the highest number of items selected and the other files are in descending order of items selected. When there is only one file in the list (that is, all the items selected come from the same file), no search of access paths is required. Otherwise, inference engine 17 is invoked to infer the access path.

Inference Engine

The first step involves searching for the optimal knowledge thread in Knowledge Base 14 to be used to generate an access path. If no optimal knowledge thread is found, the next step is to infer new knowledge threads, one of which is used to generate an access path (See FIG. 4). However, if no new threads can be inferred, then the user is informed that there is no solution.

Knowledge base 14 comprises two sections: basic and acquired knowledge threads. The basic knowledge section contains the knowledge threads that are generated by Semantics Extractor 12. The acquired knowledge section also contains knowledge threads but these are knowledge threads that have been inferred by Inference Engine 17. The process of acquiring these knowledge threads is explained later.

In this embodiment, the search is made on the acquired knowledge section first. If a thread is found where all the files in the file list using step 27 of FIG. 4 exists on the thread, this thread used to generate the access path. When the search is unsuccessful, i.e. no acquired knowledge thread is found which can match all the files in the file list, the search proceeds using the basic knowledge section. FIG. 5 shows a flow chart of one method of performing the search for a knowledge thread from the basic knowledge section to be used to generate an access path.

The following example illustrates this search. Assume the file list from Step 27 in FIG. 4 has been build up from the items the user has selected and it contains two files as follows:

File list:
EMPLOYEES
BRANCHES

Assume that the basic knowledge section has the following three basic knowledge threads with EMPLOYEES as the thread-head:

Thread 1: EMPLOYEES→BRANCHES→EXPENSES
Thread 2: EMPLOYEES→PAY
Thread 3: EMPLOYEES→BILLINGS

We define the number of files accessed as the number of files in the thread which exists in the file list starting from the thread head.

Thread 1 has files which match all those in the file list, namely EMPLOYEES and BRANCHES. The number of files accessed in this case is two (EMPLOYEES→BRANCHES) and is equal to the number of files in the file list. This thread is considered an "optimal thread" as all the files in the file list exist contiguously on the thread starting from the thread-head. Thread 2 and 3 are invalid because BRANCHES is not found on the thread.

When an optimal thread is found, the search ceases and the optimal thread is used to generate the access path. In the above example, the thread EMPLOYEES→BRANCHES→EXPENSES is used to generate the access path but only up to the BRANCHES file with the EXPENSES file ignored or "trimmed out" as it is not required in this example of the user query. Otherwise, Inference Engine 17 attempts to infer new threads by joining two or more threads together in parallel (see FIG. 4). Those new threads that are found to be optimal, i.e. they have the number of files accessed equal to the number of files in the file list from step 27 of FIG. 4, are then classified as acquired knowledge threads and stored in the acquired knowledge section of Knowledge Base 14.

The following section describes the process of deriving acquired knowledge threads. Before inferring new threads, Inference Engine 17 must first generate a list of knowledge threads consisting of basic knowledge threads which have at least one file matching any file in the file list from step 27 of FIG. 4. This thread list is then sorted in descending order of the number of files in each thread matching those in the file list. Within this sorted list of knowledge threads, if there are more than one thread with the same number of files matching those in the file list, the threads are then sorted in ascending order of the number of files in each thread. This results in a knowledge thread that has the most number of files matching those in the file list but has the least number of files on the thread being at the top of the list. This forms the sorted thread-list. This list is then used to form new threads by joining in parallel two or more threads. FIGS. 6*a* and 6*b* form a flow chart of how these new threads are inferred from the sorted thread-list.

For ease of explanation of how the above procedure works, assume that the file list from step 27 of FIG. 4 contains the following three files:

EMPLOYEES
BRANCHES
PAY

and the basic knowledge section of the Knowledge Base 14 contains the following three threads as before:

Thread 1: EMPLOYEES→BRANCHES→EXPENSES
Thread 2: EMPLOYEES→PAY
Thread 3: EMPLOYEES→BILLINGS

Thus there is no basic thread that contains all three files in the file list. Inference Engine 17 next employs parallel join inferencing. The basic knowledge threads which have at least one file found in the file list is extracted and sorted as follows:

Thread-list:	No. of files in thread matching those in file list	Thread-length
1. EMPLOYEES → PAY	2	2
2. EMPLOYEES → BRANCHES → EXPENSES	2	3
3. EMPLOYEES → BILLINGS	1	2

The flow chart of FIGS. 6*a* and 6*b* is then applied. First, the thread EMPLOYEES→PAY is added to the parallel list. Next, the file EMPLOYEES is read from the file list. Since EMPLOYEES exists on this thread it is removed from the

file list. The next file from the file list is then read and examined whether it exists in the same knowledge thread. Since it does, it is also removed. However, the next file from the file list, namely BRANCHES, does not exist in the knowledge thread. It therefore remains in the file list. The next step is to retrieve from Knowledge Base **14** the basic knowledge threads whose thread head (first file) is EMPLOYEES.

As stated earlier, the Knowledge Base contains the following basic knowledge threads:

1. EMPLOYEES→BRANCHES→EXPENSES
2. EMPLOYEES→PAY
3. EMPLOYEES→BILLINGS

The basic thread EMPLOYEES→BRANCHES→EXPENSES has the BRANCHES file and is therefore added to the parallel list. But before it is added, the EXPENSES file is removed as it is not a file in the file list. The parallel list now contains the following threads:

Parallel list:

EMPLOYEES→PAY (obtained from the sorted thread list)

EMPLOYEES→BRANCHES (obtained from the basic knowledge threads)

The above knowledge threads have a parallel relationship through the common file EMPLOYEES and form the parallel thread as follows:

```
EMPLOYEES → BRANCHES
          → PAY
```

Next, the optimality test is applied. As the number of files accessed (which is earlier defined as the number of files in the thread starting from the thread-head) is equal to the number of files in the file list, an optimal solution has been found. This optimal thread is then added to Knowledge Base **14** as an acquired knowledge thread. However, if the optimality test fails, the above process to search for new parallel relationships is then repeated using the next thread on the sorted thread-list. If there are any new parallel relationships found, the optimality test is again applied.

In the event the sorted thread-list has been exhausted with no optimal acquired knowledge thread formed from the parallel relationships found, the user is prompted to select one of the parallel relationships found if there are more than one, or else the single parallel relationship found is used to generate the access path.

In cases where there are more than one parallel relationships, there may exist parallel relationships as follows:

```
a. EMPLOYEE → PAY
            → BRANCHES
b. EMPLOYEE → BRANCHES
            → PAY
```

As both these parallel relationships are semantically the same, one is redundant and is thus removed.

There may also be a case whereby there are no parallel relationships found after the sorted thread list has been exhausted. In such a case, there is no solution to the end-user query.

The inference method as illustrated in the flow chart of FIGS. *6a* and *6b* also takes into account inference of relationships using more than one file as common files. For example, the method is able to infer the following parallel relationships, whereby EMPLOYEES and PAY are the two common files:

EMPLOYEES→PAY→file1

EMPLOYEES→PAY→file2

These then form the parallel thread:

```
EMPLOYEES → PAY → file1
                → file2
```

Program Generator

Based on the path inferred by Inference Engine **17**, the corresponding QUIZ ACCESS statement is generated. In 'QUIZ', the file linkage is specified using the ACCESS statement with the following syntax:

```
ACCESS file  {      LINK      item OF file TO item OF file }
    [ [ { AND } item OF file TO item OF file ] . . ]
        { LINK }
where ACCESS, LINK, AND, OF, TO are part of the 'QUIZ'
syntax,
      file refers to the file name,
      item refers to the item name in the file to be linked,
      [ ] means optional statement,
      { } means choose one of the options i.e. AND or LINK,
      . . means repeats one or more times
```

In QUIZ, there are two ways of defining a linkage between a number of files: hierarchical and parallel. A hierarchical linkage is defined with the "LINK . . . TO" option of the ACCESS statement. A parallel linkage is defined with the "AND . . . TO" option of the access statement.

When a single thread is used to generate the access path, the hierarchical link is used. When a combination of two or more threads are used a parallel link is used. From the earlier examples,

```
1. EMPLOYEES → BRANCHES
   uses the hierarchical linkage to generate
   ACCESS EMPLOYEES LINK Br-no OF EMPLOYEES
                TO Br-no OF BRANCHES
2. EMPLOYEES → BRANCHES
                →PAY
   uses the parallel linkage to generate
   ACCESS EMPLOYEES LINK Br-no OF EMPLOYEES
                TO Br-no OF BRANCHES
                AND Emp-no OF EMPLOYEES
                TO Emp-no OF PAY
```

It should be noted that for a different language implementation, Inference Engine **17** and Knowledge Base **14** design need not change. Only the access path needs to be rewritten using the designated language.

Compiler/Executor

After Source Program **19** is generated, Compiler/Executor **20** is used to compile Source Program **19** into executable code. The compiled program is then executed to produce the report. In this embodiment, Compiler/Executor **20** is the QUIZ part of POWERHOUSE fourth generation language.

ALTERNATIVE EMBODIMENTS

Several additional embodiments are also taught, as will now be described.

In one embodiment, a Model Purifier **26** is included which allows a user to add or alter the key type and binary relationships in Knowledge Base **14** (FIG. **7**). The Model Purifier interfaces with Semantics Extractor **12** to change Knowledge Base **14** accordingly.

In another embodiment, the functionality of Knowledge Base **14** is extended in order to store items that are derived from items in the database model. In this embodiment, the Model Purifier allows a user to input the specifications for the derived items. Derived items can also be obtained from source code of applications programs that access the database, in which case Semantics Extractor **12** serves to extract the derived item specifications from these programs. These derived items and their related database items together form a pseudo database file. Semantics Extractor **12** then uses this pseudo file to derive new binary relationships with normal database files and build new basic knowledge threads. The new binary relationships and new knowledge threads are then stored in Knowledge Base **14**. In this embodiment, the functionality of Program Generator **18** is also extended so that after Inference Engine **17** has generated the access path which may contain pseudo database files as well as normal database files, Program Generator **18** uses this access path to generate source programs to obtain information from the normal database files and the pseudo database files.

In another embodiment, another component called the Security Model Specifier **29** is included to allow a user to input a security model which specifies the items of the database model that the user can access. This is called item security. The security model also specifies the range of values within an item that the user can access. This is called value security. To support this security, the functionality of Knowledge Base **14** is extended to store the security model. Functionality of Report Item Selector **16** and Program Generator **18** are also extended to use the security model in the Knowledge Base so that the information obtained from the database at query time meets the security model specification.

In another embodiment, the functionality of the Model Purifier is extended to allow a user to specify multiple domains for a data item and the aliases for a file containing this item. In this embodiment, the functionality of Program Generator **18** is also extended to generate the appropriate file alias statements in the source program to access the database to satisfy the user query.

FIG. **7** depicts one embodiment of this invention in which Model Purifier **26** is used. Model Purifier **26** serves to allow a user to add or alter the key type of items in Knowledge Base **14**, e.g. the key type of an item can be changed from unique to repeating key. Model Purifier **26** is also used to allow a user to alter the binary relationships between database files located within Knowledge Base **14**.

The rationale for the use of Model Purifier **26** in accordance with this embodiment of the present invention is that in some applications the database model or the application programs that access the database may not be rich enough for Semantics Extractor **12** to extract the necessary semantics including file linkages for the user to perform certain queries. Model Purifier **26** allow the user to input the additional semantics to satisfy these queries.

FIG. **8** shows one embodiment of this invention in which Semantics Extractor **12** is extended to interface with Model Purifier **26**. If the key types are altered, Model Purifier **26** activates Semantics Extractor **26** to re-derive the binary relationships and to rebuild the knowledge threads in Knowledge Base **14**. If the binary relationships are altered, Model Purifier **26** activates Semantics Extractor **14** to rebuild the knowledge threads.

A user may specify an item or items to be reported that may not be found in Database Model **10**. Examples of such items are as follows, which we called derived items as they are obtained by defining using the items of the database files:

	Derived Item	Defined as
(i)	Employee-Name	Firstname of Employee file + Lastname of Employee file
Note:	Firstname & Lastname are items of database file Employee	
(ii)	Sales-Commission	Sales-Amount of Invoice file × Commission Rate of Commission Table file
Note:	Sales-Account and Commission Rate are items of database file Invoice and Commission-Table, respectively	
(iii)	Total-Sales	sum of Sales-Amount of Invoice file for each month of the year
(iv)	Ratio-of-Jan-Sales to-Total-Sales	(Sales-Amount of Invoice file for January)÷(Total-Sales derived from (iii) above)

These derived items can be obtained from direct user input into Knowledge Base **14** or from Source Code of Application Programs **27** that access the database. To obtain such derived items from direct user input, the functionality of Model Purifier **26** is extended to meet this requirement. To obtain such derived items from Applications Programs **27**, the functionality of Semantics Extractor **12** is extended with Program Analyzer **28** (see FIG. **9**) to extract the derived items from these programs. In addition to the definitions of the derived items, the data type, size, and format of these derived items are also extracted from Applications Programs **27** by Program Analyzer **28** of Semantics Extractor **12**. These derived items and their related database items together form a pseudo database file which is included in File Definition List **22** (see FIG. **9**). Semantics Extractor **12** then uses this pseudo file in File Definition List **22** to derive new binary relationships with normal database files in File Definition List **22** and build new basic knowledge threads. The new binary relationships and new knowledge threads are stored in Knowledge Base **14**. Besides the Model Purifier **26** and Semantics Extractor **12**, the Program Generator **18** is also extended so that after the Inference Engine has generated the access path which may contain pseudo database files as well as normal database files, the Program Generator **18** uses this access path to generate source programs to obtain information from the normal database files and pseudo database files.

The following is a description of one embodiment of an algorithm suitable for use by Program Analyzer **28** of Semantic Extractor **12** to extract the derived items in accordance with the teachings of this invention:

a. From a single pass application program

A single pass application program is a program that contains only one database access statement.

Step 1: Extract the database files that are accessed in the application program.
  For each database file accessed
    Extract the list of items of database file.
    Store this list and the database file name as a pseudo database file in the knowledge base.
    Next database file

Step 2: Extract the list of derived items and their definitions from the application program
  For each derived item
    Scan for the data type, size and format. Store the derived item name, its definition, data type, size and format in the same pseudo database file obtained from step 1 above.
  Next derived item

b. From a multiple pass application program

A multiple pass application program consists of many single pass application program "stringed" together with the output of the previous pass being used as input by the current pass.

Step 1: Create a new pseudo database file 1 for the first pass (i=1) using the steps of the single pass application program above.

Step 2: For each of the remaining passes i.e. i=2 to n with n being the last pass,

Create a pseudo database file i

Store the location of the previous database pseudo file (i−1) in pseudo database file i.

These derived items are then presented to the user together with the items from the database model for the user to select from in order to satisfy the user query.

In order to generate a program from the access path inferred by Inference Engine 17 that may contain pseudo database files, in addition to the above described extension of the functionality of Model Purifier 26 and Semantics Extractor 12, the functionality of Program Generator 18 is also extended as follows:

Step 1: Examine the access path generated by the Inference Engine 17.

Step 2: For every file in the access path:

If it is a pseudo database file,

Then

(i) generate source program to produce data for the pseudo database file using the derived item definitions; and

(ii) generate source program to extract the information from both the database and the data of the pseudo database file produced by the program of step 2 (i) above.

Next file

In another embodiment, a security model specifies the items a user can access (item security) and the range of values within an item the user can access (value security). In item security, a user is assigned access rights to a subset of the list of items available in Knowledge Base 14. In value security, a user is assigned access rights to a range or ranges of values within an item in Knowledge Base 14. For example, the access rights could be DEPT-NO=101, SALARY <2000, or SALARY of PAY file if EMP-GRADE of EMPLOYEE file >8.

FIG. 10 shows the Main Flowchart of one embodiment of the present invention which supports a security model. As shown in FIG. 10, Security Model Specifier 29 is provided for the user to input the security model into Knowledge Base 14. Also, in accordance with this embodiment, the functionality of Report Item Selector 16 and Generate File List 27 module (FIG. 4) of Information Scout 15, as well as Program Generator 18 are extended to support the security model as is now explained.

In one embodiment, the functionality of Report Item Selector 16 is extended such that only the items defined in the security model that are accessible by the user will be presented to the user for selection at query time.

In one embodiment, the functionality of Generate File List module 27 is extended to perform the following:

Step 1:

For each query item selected Retrieve the file(s) containing the query item.

Next query item

Step 2:

For each file retrieved in step 1

For each item in the file

Retrieve any value security defined

Next item

Next file

Step 3:

For each value security retrieved in steps 1 and 2

If the security definition involves value from another file

Then add the file to the file list for inferring the access path

Next value security

In one embodiment of this invention, the functionality of Program Generator 18 is extended to perform the following:

Step 1:

For each query item selected

Retrieve any value security defined on the item.

Retrieve the file(s) containing the query item.

Next query item

Step 2:

For each file retrieved in step 1

For each item in the file

Retrieve any value security defined

Next item

Next file

Step 3:

For each value security retrieved in step 2

Join the value security definition using the AND condition.

Next value security

In yet another embodiment of this invention, the functionality of the Model Purifier is extended to specify multiple domains for a data item. An example of an item with multiple domains is Code Id of a database file called Master Code which may contain 2 items, namely Code Id and Code Description. The data for the Master Code file could, by way of example, be as follows:

Code Id	Code Description
RACE01	Chinese
RACE02	Caucasian
.	
.	
.	
CITY01	San Francisco
CITY02	Singapore
.	
.	
.	

In this example, this file is used to store codes for races and cities. However, races and cities are two separate domains. To break the codes into the two separate domains, the Code Id must be redefine as Race Code and City Code. The Master Code file will then have the following file aliases:

Race Master Code to contain Race Code Id

City Master Code to contain City Code Id

To support this multiple domain for an item, we need to extend the Model Purifier to allow a user to specify the multiple domains and the corresponding file aliases. FIGS. 12 and 13 depict one embodiment of a main flowchart and Semantics Extractor 12 of the present invention which are capable of supporting multiple domains. The redefined items, eg Race Code Id and City Code Id, are stored in Keyword Library 13 and the alias file(s) are stored in Knowledge Base 14. In this embodiment, the functionality of Program Generator 18 is extended to identify the alias file(s) in the access path generated by Inference Engine 17 and to generate the necessary alias file(s) statements in the source program.

There are also a number of alternative embodiments of this invention related to how the Knowledge Base is physically implemented. The various forms of implementations can be considered in the following manner:

a. Precreated versus Run-time Creation

b. Permanent Storage versus Temporary Storage

c. Outside the Database Model versus Inside the Database Model

In precreated mode, the Knowledge Base is created once and used for every query over a period of time. Should the Database Model change, then the Knowledge Base is recreated to properly reflect the changed Database Model.

In run-time mode, the Knowledge Base is created at query time for every query, regardless of whether the Database Model has or has not changed since a previous query.

In permanent storage, the Knowledge Base is usually implemented in secondary storage devices such as a disk drives. In temporary storage the Knowledge Base is implemented in memory such as RAM (random access memory). A Precreated Knowledge Base is preferably (although not necessarily) implemented in permanent storage while a Run-Time Knowledge Base is preferably (although not necessarily) implemented in temporary storage. The reason is that as precreated knowledge is created once and reused many times, it would be beneficial to keep it permanent so that even if the electrical power to the computer system is switched off, the Knowledge Base is retained and can be used again once power is restored. In run-time mode, as the Knowledge Base is created at query time and not reused for the next query, it is not necessary to implement the Knowledge Base in permanent storage.

In POWERHOUSE fourth generation language, the database model is implemented in the data dictionary. In SQL database language, the database model is implemented in the system catalog.

In one embodiment of the present invention, the Knowledge Base is implemented outside the data dictionary or system catalog. In an alternative embodiment of the present invention, the Knowledge Base is implemented to reside partially or fully inside the data dictionary or system catalog, as the data dictionary or system catalog is a form of storage that can contain the Knowledge Base.

GENERATION OF CLASSES AND THEIR TRANSLATION INTO ENTITY-RELATIONSHIP MODELS

An additional embodiment is taught as will now be described. The purpose of this embodiment is to provide another way for a user to interface with Report Item Selector **16** to formulate this query. Instead of a long list of items for a user to pick as described earlier the user is initially prompted for a class to pick out of a number of classes that make up his application. An entity-relationship (E-R) model of the selected class is then presented to the user for selection of the desired attributes. The desired attributes then constitute a designation of the information to be extracted from the database. It is also possible for the user to formulate the query by selecting the desired attributes from two different classes.

This invention introduces a way of selectively grouping the database files into what we called classes and translating each class definition into an E-R model. These classes represent the different types of high-level objects that make up the application, e.g. employees, customers, orders, invoices, etc.

This embodiment, as shown in FIG. **11**, involves the following:

the extension of Semantics Extractor **12** to extract more semantics from Database Model **10**

the addition of Class Generator **30**

the addition of E-R Model Translator **31**

the addition of E-R Model of Classes File **32**

Extension of Semantics Extractor **12**

FIG. **12** is a flowchart of one embodiment of this invention utilizing Semantics Extractor **12** which is extended to support the extraction of additional semantics from the Database Model **10**. There are several additional parts to this embodiment.

Classify Binary Relationships (BR)

Classify Binary Relationships step **33** classifies each binary relationship in Binary Relationships File **24** into one of the following types, in this embodiment:

has_children

has_wards

has_subtype

We shall now describe these three different types of binary relationships and explain how they can be identified from the key types of the items of the database files.

Earlier we have described six kinds of file linkages, namely

UR (unique key to repeating key)

RU (repeating key to unique key)

NU (non-key to unique key)

UU (unique key to unique key)

RR (Repeating key to repeating key)

NR (non-key to repeating key)

Of these six kinds, both the RR and NR combination should not exist in a normalized data model as they represent a bad file design.

Let us now consider the remaining four kinds of file linkages. For the UR kind, there are two possible types of binary relationships that can exist, namely

has_children

has_wards

Let us explain the meaning of these two types with examples. Suppose we have an EMPLOYEE file with Emp_No as a unique key and another file called SKILLS to contain the skills of every employee. The SKILLS file has a repeating key item called Emp_No and a non-key item called Skill but no unique key. The binary relationship between these two files is as follows:

Source File	Item	Target File	Item	Relationship
EMPLOYEE	Emp_No	SKILLS	Emp_No	UR

This binary relationships is a "has_children" type because EMPLOYEE is not only related to SKILLS but the relationship is one where EMPLOYEE considers SKILLS as its "children". This is because the records of SKILLS can only be created if their corresponding EMPLOYEE record exists. This "has_children" type of UR binary relationship can be identified by the fact that the source file of the binary relationship has a unique key but the target file has a repeating key but no unique key.

Let us consider another example involving 3 files as follows:

File	Unique Key	Repeating Key
CHAPTER	Chap_No	—
SECTION	Chap_No, Sect_No	Chap_No
PARAGRAPH	Chap_No, Sect_No, Para_no	Chap_No, Sect_No

In this case CHAPTER has a unique key called Chap_No and no repeating key. SECTION has a composite unique key and a repeating key within the composite unique key. These files have the following binary relationships:

Source File	Item	Target File	Item	Relationship
CHAPTER	Chap_No	SECTION	Chap_No	UR
SECTION	Chap_No, Sect_No	PARAGRAPH	Chap_No, Sect_No	UR

Both these binary relationships are "has_children" type because records of SECTION can only EXIST if the corresponding record of CHAPTER exists. Similarly, records of PARAGRAPH can only exist if the corresponding record of SECTION exist. These "has_children" type of UR binary relationships can be identified by the fact that the source file and target file have a unique key and the target item is a repeating key within the unique key of the target file as well as the binary relationship being UR.

Consider another example where a CUSTOMER file has Cust_No as its unique key and another file INVOICES has Inv_No as its unique key and Cust_No as its repeating key. They both have a binary relationship as follows:

Source File	Item	Target File	Item	Relationship
CUSTOMERS	Cust_No	INVOICES	Cust_No	UR

This binary relationship is a "has_wards" type. In this case, we cannot consider CUSTOMER as having INVOICES as its "children" since INVOICES have their own identity through their own unique key, namely Inv_No. Instead, we could consider CUSTOMER as a "guardian" having INVOICES as its "wards" because INVOICES belong to their respective CUSTOMER.

A "has_wards" type can be identified by the fact that both the source and target files have a unique key and the target item is not part of the target file unique key as well as the binary relationship being UR.

Let us next consider the NU and RU binary relationships. The RU binary relationships are the inverse of the UR Binary relationships. We therefore classify them as either the "inverse_of_has_children" or the "inverse_of_has_wards". As for the NU binary relationships, we create an inverse of it and assign to this inverse a "has_wards" type. We then store it in Knowledge Base **14**. For example, suppose we have an EMPLOYEE file having a NU binary relationship with a BRANCHES file using the Branch_Code item from both files as follows:

Source File	Item	Target File	Item	Relationship
EMPLOYEE	Branch_Code	BRANCHES	Branch-Code	NU

We create an inverse as follows with the type specified as "has_wards":

Source File	Item	Target File	Item	Type
BRANCHES	Branch_Code	EMPLOYEE	Branch_Code	has_wards

This binary relationship is then stored in Knowledge Base **14**.

Lastly let us consider the UU binary relationship. Such a binary relationship is called a "has_subtype" type. Consider for example, a file called EMPLOYEE with Emp_No as its unique key and two other files MONTHLY_RATED_EMP and DAILY_RATED_EMP both of which also have Emp_No as their unique key. As an example, an employee is either a monthly rated employee or a daily rated employee but not both. We consider this as EMPLOYEE having MONTHLY_RATED_EMP and DAILY_RATED_EMP as its subtypes. The following binary relationships between EMPLOYEE and the two other files with EMPLOYEE as the source file reflect this "has_subtype" type of binary relationships:

Source File	Item	Target File	Item	Type
EMPLOYEE	Emp_No	MONTHLY_RATED_EMP	Emp_No	UU
EMPLOYEE	Emp_No	DAILY_RATED_EMP	Emp_No	UU

If a file has more than one subtype it is possible to automatically identify which of the two opposite UU binary relationships is a "has_subtype" and which the "inverse of has subtype" by comparing them with another pair of opposite UU binary relationships. The ones with the same source file are then the "has_subtype". However, it is not possible to do so if a file has only one subtype. In a later section of this patent description, we explain that a user will have to input this knowledge using Model Purifier **26**. (Note: In the SQL language using primary keys and foreign keys, it is possible to automatically identify which of the two opposite binary relationships is a "has_subtype" even if a file has only one subtype.)

We earlier describe the extension of Model Purifier **26** to allow an user to manually specify multiple domain and corresponding file aliases. In this embodiment, this process can now be automated using the binary relationships. We shall now describe how the procedure Derive Binary Relationship **23** can be further extended to identify files that requires file aliases to be defined and to define these file aliases. A file that require file aliases to be defined can be identified by the fact that there are more than one binary relationships of the NU or RU kind which has the same target file and item. For example, suppose there are two NU binary relationships as follows:

Source File	Item	Target File	Item	Type
EMPLOYEES	Race_Code	MASTER_CODE	Code	NU
EMPLOYEES	Citizenship_Code	MASTER_CODE	Code	NU

The target file, namely MASTER_CODE, and target item, namely Code, is the same for both binary relationships. For each such binary relationships, we create a file alias of the target file and replace the target file of the binary relationship with the file alias, e.g. for the above two binary relationships, we create the following file aliases for the MASTER_CODE file and store them in the Knowledge Base **14**:

EMPLOYEE_Race_Code

EMPLOYEE_Citizenship_Code and change the binary relationships to:

Source File	Item	Target File	Item	Type
EMPLOYEE	Race_Code	EMPLOYEES_Race_Code	Code	NU
EMPLOYEE	Citizenship_Code	EMPLOYEES_Citizenship_Code	Code	NU

Identify Entity Type of Each File

In Identify Entity Type of Each File step **34** a database file is classified as one of the following types:

kernel entity

characteristic entity

associative entity

subtype entity

pure lookup entity

Kernel entities are entities that have independent existence, they are "what the database is really all about". In other words, kernels are entities that are neither characteristic nor associative, e.g. suppliers, parts, employees, orders, etc. are all kernel entities.

A characteristic entity is one whose primary purpose is to describe or "characterize" some other entity. For example, the file SKILLS which contains the SKILLS an employee has is a characteristic entity of the EMPLOYEE entity. Characteristic entities are existence-dependent on the entity they described which can be kernel, characteristic, or associative.

An associative entity is an entity whose function is to represent a many-to-many (or many-to-many-to-many, etc.) relationship among two or more other entities. For example, a shipment is an association between a supplier and a part. The entities associated may each be kernel, characteristic, or associative.

A subtype is a specialization of its supertype. For example, as described earlier MONTHLY_RATED_EMP and DAILY_RATE_EMP are subtypes of EMPLOYEE.

Lastly, we have entities that look like kernel entities but should not be classified as one because their purpose is solely for lookup of code description.

FIG. **13***a* describes the procedure to identify kernel and pure lookup entities. It first identifies those database files that are either kernel or pure lookup entities using the following rule:

1) Identify those database files that have a unique key.

2) Of these database files, eliminate those that are used as a target file in any "has_children" or "has_subtype" binary relationships.

These database files are either kernel or pure lookup entities. To distinguish between the two it next uses the following rule:

1) IF such a database file has no "children" or "sub-type", i.e. it is not a source file in any "has_children" or "has_subtype" binary relationship; AND

2) IF it is not a "ward" i.e., it is not a target file in any "has_wards" binary relationship;

3) THEN it is a pure lookup entity;

4) OTHERWISE it is a kernel entity.

FIG. **13***b* describes one embodiment of a procedure to identify characteristic and associative entities. It uses the following rule:

1) IF a database file appears more than once as a target file in "has_children" binary relationships, THEN it is an associative entity;

2) OTHERWISE, IF it appears only once, THEN it is a characteristic entity.

Subtype entities are easily identified as they are the target files in "has_subtype" binary relationships.

Note that the SYSFILES as used in FIG. **13***a* is a part of File Definition List **22** which has been redefined as consisting of two files, namely

SYSFILES

SYSFILEITEMS

SYSFILES is used to store the following:

file name

indicator whether it is an alias file or a real file having alias files

if alias file, its real file name

entity type

SYSFILEITEMS is used to store the following:

file name

item name

item type (e.g. character, numeric, date)

keytype of item, e.g. unique key, repeating key, or non-key

Reclassify Certain Entities and Binary Relationship **35**

Even though we have earlier identified some files as kernel entities, some of these kernel entities should be reclassified as pure lookup entities. Consider for example a kernel entity EMPLOYEE having an associative entity LANGUAGE_SPOKEN whose other kernel entity is LANGUAGE. LANGUAGE_SPOKEN has two items, namely a repeating key called Emp_No and another repeating key called Language_Code. The LANGUAGE file has only two items, namely a unique key item called Language_Code and a non-key item called Language_Desc. Even though we earlier identified LANGUAGE as a kernel entity since it has LANGUAGE_SPOKEN as its "children", LANGUAGE should be reclassified as a pure lookup entity since it is used solely by the LANGUAGE_SPOKEN file to obtain the description of the Language_Code.

FIG. **14** is a flow chart depicting one embodiment of a procedure to identify these kernel entities and modify them to pure lookup entities. It uses the following rule to do this:

1) IF a kernel entity has only associative entities and no characteristic entities; and

2) IF it is not a target file in any "has_ward" binary relationships;

3) THEN modify the kernel entity into a pure lookup entity. Also, modify the associative entities of this kernal entity into characteristic entities if the associative entities has only one other entity that associates it.

Next we access all those "has_children" and "has_wards" binary relationships whose source file is one of these pure lookup entities. We then modify them into a new type called "inverse_of_pure lookup" type. We use the word "inverse" as the lookup direction is not from source file to target file but from target file to source file.

Class Generator **30**

Class Generator **30** (FIG. **11**) generates a definition of a class for each kernel entity in the database and stores this definition as a Class Definition File (CDF) in Knowledge Base **14**. A class is a cluster of files whose structure is a tree. The root of the tree is a kernel entity which defines the core attributes of the class. The tree has the following main branches:

(i) a branch for each of the subtypes of the root kernel entity

(ii) a branch for each of the "wards" of the root kernel entity

(iii) a branch for each of the characteristic entities of the root kernel entity

(iv) a branch for each of the associative entities of the root kernel entity

These branches are derived using the "has_subtype", "has_wards", and "has_children" binary relationships with the root kernel entity being the source file.

Each of the above subtype, characteristic, and associative entities could also have their own branches which are their characteristic or associative entities. The latter characteristic or associative entities could also have their own characteristic or associative entities, and so forth. These branches are derived using the "has_children" binary relationship with the target files of these binary relationships forming the new branches. The procedure of FIG. **15***a* and **15***b* together with the sub-procedure Include_File (list) of FIG. **16***a* and **16***b* are used to derive the above branches which are then stored as a set of lists in Class Definition File A (CDF A). An example of such a list is:

File=EMPLOYEE
item=Emp_No
File=BILLINGS
item=Emp_No
BR Type="has_children"
item=Proj_No
File=PROJECTS
item=Proj_No
BR type="inv_of_has_children"

This list contains a file EMPLOYEE, linked to a file BILLINGS, which is linked to file PROJECTS. The binary relationship (BR) type from EMPLOYEE to BILLINGS is "has_children" using the item Emp_No from both files and the BR type from BILLINGS to PROJECTS is "inv_of_has_children" using the item Proj_No from both files.

Let us now describe how a list in CDF A is produced using this procedure. It starts by initializing a list to the first kernel entity in SYSFILES. This kernel entity forms the root kernel entity of a class to be generated. Next it looks for a subtype entity of this kernel entity using a sorted SYSFILES and the Binary Relationships File **24**. This file has been sorted in descending order of subtypes, kernels, associative, characteristics, and pure lookups. If one subtype entity is found, it is added to the list together with the name of the corresponding binary relationship type, which in this case is "has_subtype" and the names of the items used. It then calls on a sub-procedure Include_File (list), for example as depicted in FIG. **16***a* and **16***b*, to find associative and characteristic entities of the subtype entity. If a characteristic entity is found, it is added to the list together with the name of the corresponding binary relationship type which in this case is "has_children" and the names of the items used. The sub-procedure Include_File (list) is then called again, this time to find other associative entities or characteristic entities of the characteristic entity. If no such entity can be found the list is then written to the Class Definition File A (CDF A).

Besides these branches, the tree of a class also has what we called "lookup" branches originating from each node in the above branches. These "lookup" branches are derived using the "inv_of_has_wards" and "pure_lookup" binary relationships, with the node being the source file and the target file forming new branches. Furthermore, the new branches could also have their own new "lookup" branches and so forth. These subsequent "lookup" branches are formed using not only the "inv_of_has_wards" and "pure_lookup" binary relationships but also the "inv_of_has_children" binary relationships with the target file forming the new "lookup" branches. The procedure of FIG. **17** together with the sub-procedure Process_File of FIGS. **1***a* and **18***b* are suitable for use to derive these branches, which are then stored as a set of lists in Class Definition File B (CDF B).

Let us describe one example of how a list in CDF B is produced using this procedure. It starts by reading in the first record of CDF A. A list is then initialized to the first file in this CDF A record. A check is made to see if this file has been processed before. Since it is not, the sub-procedure Process_File (list), for example as depicted in FIG. **18***a* and **18***b*, is then called to look for a "lookup" file for this file. If there is such a file, it is added to the lists together with the corresponding binary relationship type and the sub-procedure is called again. If no further "lookup" file can be found, the list is written to CDF B.

Let us now apply the above procedures on the exemplary personnel system below to generate the classes for this system.

File	Items	Item Type	Key Type
EMPLOYEE	Emp_No	Numeric	Unique key
	Emp_Name	Character	Non-key
	Branch_No	Numeric	Non-key
	Race_Code	Character	Non-key
	Address	Character	Non-key
	Salary	Numeric	Non-key
MANAGER	Emp_No	Numeric	Unique key
	Co_Car_No	Character	Non-key
NON_MANAGER	Emp_No	Numeric	Unique key
	Union_M_No	Character	Non-key
BRANCH	Branch_No	Numeric	Unique key
	Branch_Name	Character	Non-key
	Br_Tot_Expenses	Numeric	Non-key
	Country_No	Numeric	Non-key
RACE_CODE	Race_Code	Character	Unique key
	Race_Desc	Character	Non-key
SKILLS	Emp_No	Numeric	Repeating key
	Skill	Character	Non-key
PROJECT	Proj_No	Numeric	Unique key
	Proj_Name	Character	Non key
	Cust_No	Numeric	Non-key
BILLINGS	Emp_No	Numeric	Repeating key
	Proj_No	Numeric	Repeating key
	Month	Date	Non-key
	Amount	Numeric	Non-key
CUSTOMER	Cust_No	Numeric	Unique key
	Cust_Name	Character	Non-key
EXPENSES	Branch_No	Numeric	Repeating key
	Month	Date	Non-key
	Adv_Exp	Numeric	Non-key
	Pers_Exp	Numeric	Non-key
COUNTRY	Country_No	Numeric	Unique key
	Country_Name	Character	Non-key

Using extended Semantics Extractor **12** of FIGS. **11** and **12**, the following binary relationships are derived:

Source Film	Item	Target File	Item	Type
EMPLOYEE	Emp_No	SKILLS	Emp_No	has_children
EMPLOYEE	Emp_No	BILLINGS	Emp_No	has_children

-continued

Source Film	Item	Target File	Item	Type
EMPLOYEE	Emp_No	MANAGER	Emp_No	has_subtype
EMPLOYEE	Emp_No	NON_MANAGER	Emp_No	has_subtype
BRANCH	Branch_No	EMPLOYEE	Branch_No	has_ward
BRANCH	Branch_No	EXPENSES	Branch_No	has_children
COUNTRY	Country_No	BRANCH	Country_No	inv. of pure_lookup
RACE_CODE	Race_Code	EMPLOYEE	Race_Code	inv. of pure_lookup
PROJECT	Proj_No	BILLINGS	Proj_No	has_children
CUSTOMER	Cust_No	PROJECT	Cust_No	inv. of pure_lookup

Also, the entity type of each file in SYSFILES is as follows:

File	Entity Type	Short Name of File
EMPLOYEE	Kernel	K1
MANAGER	Subtype	S1
NON-MANAGER	Subtype	S2
BRANCH	Kernel	K2
RACE_CODE	Pure Lookup	L1
SKILLS	Characteristic	C1
PROJECT	Kernel	K3
BILLINGS	Associative	A1
CUSTOMER	Pure Lookup	L2
EXPENSES	Characteristic	C2
COUNTRY	Pure Lookup	L3

Let us first apply the exemplary procedure of FIGS. 15*a* and 15*b* on the personnel system example. It first initializes a list to the first kernel entity in SYSFILES which is EMPLOYEE (K1). This means that it is going to generate Class Definition File A for the EMPLOYEE class. Next it uses a sorted SYSFILES and the Binary Relationship File 24 to find other entities to add to this list. The sorted SYSFILES contains files in descending order of subtypes, kernels, associatives, characteristics and pure lookups. The next entity added to the list is S1 which is MANAGER, with the items being Emp_No and the BR type being "has_subtype". This list is then written to CDF A.

The list is initialized again to K1 and S2 added to it, with the items being Emp_No and the BR type being "has_subtype", after which it is written to CDF A.

After the subtypes have been processed, the procedure initializes list K1 and search for those kernel entities that are "wards" of K1. However, K1 has no "wards" and so there are no such entities to add to the list.

Next the procedure searches for associative entities of K1. K1 has one associative entity, namely BILLINGS (A1) so A1 is added to the list, with the items being Emp_No and the BR type being "has_children". Next the procedure includes PROJECTS (K3) in this list as it constitutes the other entity that associates A1, with the items being Proj_No and the BR type being "inv_of_has_children". This list containing K1, A1, K3 is then written to CDF A.

After this, the procedure initializes the list to K1 again and search for characteristic entities of K1. K1 has one characteristic entity, namely SKILLS (C1), so C1 is added to the list with the items being Emp_No and the BR type being "has_children". The list is then written to CDF A.

At this point CDF A contains the following lists:

a list having K1, S1

a list having K1, S2

a list having K1, A1, K3

a list having K1, C1

The procedure next generates the lists for the next kernel entity, namely BRANCHES (K2). K2 has no subtype and associative entities but it has (EMPLOYEE) K1 as its "ward". This produces the list K2, K1, with the items being Branch_No and the BR type being "has_wards". This list is written to CDF A. It also has EXPENSES (C2) as its characteristic entity. This produces the list K2, C2, with the items being Branch_No and the BR type being "has_children", which is also written to CDF A.

Finally, the list for the third and last kernel entity, namely PROJECTS (K3), is produced. However, there is only one list, namely

a list having K3, A1, K1

since PROJECTS has an associative entity only which is BRANCH (A1), with K1 being the other entity that associates A1. In this list the items for K3 and A1 is Proj_No with the BR type being "has_children". The items for A1 and K1 is Emp_No with the BR type being "inv_of_has_children".

Let us next apply the exemplary procedure of FIG. 17 on the personnel system example. It uses the lists of CDF A derived earlier to generate the lists for CDF B. Using the same personnel system it first initializes a list to the first entity in the first list of CDF A, namely K1. It then adds to this list entities that are lookup entities to K1. A lookup entity is a target file in a "pure_lookup" or "inv_of_has_ward" binary relationships, with K1 as the source file. K1 looks up on BRANCH (K2), so K2 is added to the list with the items being Branch_No and the BR type being "inv_of_has_wards". Next a check is made on K2 to see if it too has lookup entities. K2 in fact has one, namely COUNTRY (L3). L3 is therefore added to the list with the items being Country_No and the BR type being "pure_lookup". At this point the list contains K1, K2, L3. Since L3 has no lookup entities, this list is written to the CDF B. Next the procedure returns to K2 to see if K2 has other lookup entities. Since it does not, the procedure returns to K1. It finds that K1 has another lookup entity, namely RACE_CODE (L1). The list containing K1, L3 with the items being Race_Code and the BR type being "pure_lookup", is then written to CDF B.

The next entity in the first list of CDF A is next processed. This entity is MANAGER (S1). It, however, does not have any lookup entities and so it is ignored.

As there is no more entity on the first list of CDF A, the first entity in the second list of CDF A is considered for processing. A check is made first to see if this entity has already been processed earlier. This entity is K1 which has been processed earlier and so it is ignored. The next entity is NON_MANAGER (S2) which has not being processed. However, it does not have any lookup entities and so it is ignored.

The above procedure is again applied for the next list of CDF A, which contains K1, A1, K3. As K1 has already been processed and A1 has no lookup entity, no list is produced for either of them. However, K3 (PROJECT) has a lookup entity, namely CUSTOMERS (L2). So the list K3, L2 is produced with the items being Cust_No and the BR type being "pure_lookup". It is written to CDF B. The next list of CDF A is K1, C1. However, since K1 has already been processed and C1 (SKILLS) has no lookup entities, both are ignored.

The above procedure is applied to the remaining lists in CDF A. In this example, only one list is produced, contain-

ing K2, L3 with the items being Country_No and the BR type being "pure_lookup".

Besides CDF A and CDF B, the Class Definition File also includes CDF C. CDF C includes a single list which contains all the pure lookup entities.

The CDF C for the personnel system example contains L1, L2, L3.

E-R Model Translator

E-R Model Translator 31 (FIG. 11) is used to produce an Entity-Relationship (E-R) model for each class using CDF A, CDF B and Binary Relationships File 24. Many different embodiments of an E-R model can be produced. The following describes one example of a procedure to produce one embodiment of an entity-relationship (E-R) model of a class for all the classes except the class containing the pure lookup entities. This procedure begins by creating another file identical to SYSFILEITEMS. This duplicate file is called TEMPFILEITEMS. Next, for each "inverse_of_pure_lookup" binary relationships in Binary Relationship File 24, it inserts all the items of the file used as source file in this binary relationship except those items of the file that are used as source items into this TEMPFILEITEMS at the point which corresponds to the target items of this binary relationship. Next, for each "has_children", "has_wards", or "has_subtype" type of binary relationships in Binary Relationship File 24, it deletes those items in TEMPFILEITEMS that are used as target items in such binary relationships.

The procedure then gradually builds the E-R model for each class making use of the resultant TEMPFILEITEMS. It first include all the items of the root kernel entity of the class into the E-R model of the class. These items are obtained from TEMPFILEITEMS. Next it applies the following Decision Tables 1 and 2 on the CDF A and CDF B of the class to determine the relationship names between two adjacent entities in the class which are not pure lookup entities.

DECISION TABLE 1

To define relationship names between two adjacent entities in the CDF A lists

Rule	From Entity	To Entity	BR Type	Relationship Name
1	K	S	has_subtype	K is a S
2	K	S	has_wards	K has S
3	K	K'	has_wards	K has K'
4	X	C	has_children	X has C
5	X	A	has_children	a. X has Y b. X and Y have A (Y is another entity that associates A)

DECISON TABLE 2

To define relationship names between two adjacent entities in the CDF B lists

Rule	From Entity	To Entity	BR Type	Relationship Name
1	X	C or A	inv_of_has_wards	X references C or A
2	K'	K	inv_of_has_wards	K' belongs to K
3	C or A	K'	inv_of_has_wards	C or A references K
4	C or A	K	inv_of_has_children	C or A belongs to K

Legend:
X - kernel, subtype, characteristic, or associative entity
C - characteristic entity
A - associative entity
K,K' - kernel entities
Y - other entity that associates associative entity

For each relationship name identified using the above tables, the relationship name and the items of the file corresponding to the second entity of the two adjacent entities that establish this relationship name are included in the E-R model of the class. The items included are obtained from TEMPFILEITEMS. This E-R model is then stored in the E-R Model of Class File 32.

Let us now illustrate this procedure by applying it on the exemplary personnel system described earlier. We know at this stage that this personnel system has three classes which are derived from the kernel entity of EMPLOYEE, BRANCH, and PROJECT. Let us call these three classes:

ABOUT EMPLOYEE

ABOUT BRANCH

ABOUT PROJECT

If we were to apply the procedure, we should get the following E-R models for the three classes:

ABOUT EMPLOYEE

- Emp_No
- Emp_Name
- Race_Code
- Race_Desc
- Address
- Salary
- <EMPLOYEE belongs to BRANCH>
  - Branch_No
  - Branch_Name
  - Br_Tot_Expenses
  - Country_Code
  - Country_Name
- <EMPLOYEE is a MANAGER>
  - Co_Car_No
- <EMPLOYEE is a NON_MANAGER>
  - Union_M_No
- <EMPLOYEE has PROJECT>
  - Proj_No
  - Proj_Name
  - Cust_No
  - Cust_Name
- <EMPLOYEE and PROJECT have BILLINGS>
  - Month
  - Amount
- <EMPLOYEE has SKILLS>
  - Skill

ABOUT BRANCH

- Branch_No
- Branch_Name
- BR_Tot_Expenses
- Country_Code
- Country_Name
- <BRANCH has EMPLOYEE>
  - Emp_No
  - Emp_Name
  - Race_Code
  - Race_Desc
  - Address
  - Salary
- <BRANCH has EXPENSES>
  - Month
  - Adv_Exp
  - Pers_Exp

ABOUT PROJECT

- Proj_No
- Proj_Name
- Cust_No
- Cust_Name
- <PROJECT has EMPLOYEE>
  - Emp_No
  - Emp_Name
  - Race_Code
  - Race_Desc
  - Address
  - Salary
- <EMPLOYEE belongs to BRANCH>
  - Branch_No

-continued

Branch_Name
Br_Tot_Expenses
Country_Code
Country_Name
<PROJECT and EMPLOYEE have BILLINGS>
Month
Amount

Let us now explain how these E-R models are produced when the procedure is applied. First another file identical to SYSFILEITEMS is created. This file is called TEMPFILEITEMS. For each "inverse_of_pure_lookup" binary relationship in Binary Relationship File **24**, the procedure inserts in TEMPFILEITEMS, at the point corresponding to the target item, all the items of the source file except the source item itself, e.g. there is an "inverse_of_pure lookup" binary relationship as follows.

Source File	Item	Target File	Item
RACE_CODE	Race_Code	EMPLOYEE	Race_Code

The items of the source file, namely RACE_CODE, are Race_Code and Race_Desc. The procedure inserts only the item Race_Desc (leaving out Race_Code as it is a source item) at Race_Code of EMPLOYEE of TEMPFILEITEMS, so that the items of EMPLOYEE in TEMPFILEITEMS becomes:

Emp_No
Emp_Name
Branch_No
Race_Code
Race_Desc
Address
Salary

Next the procedure deletes those items in TEMPFILEITEMS that correspond to the target item in "has_children", "has_wards" and "has_subtype" binary relationships in Binary Relationship File **24**. For example, Branch_No of EMPLOYEE in the exemplary personnel system is a target item in a "has_wards" binary relationship with BRANCH. Thus, this item in the TEMPFILEITEMS file is deleted.

The resultant TEMPFILEITEMS file is then used together with the two decision tables earlier described to produce the above E-R model for each class which is then stored in the E-R Model of Classes File **32**.

Let us show how the procedure produces the E-R model for the ABOUT EMPLOYEE class. First all items of EMPLOYEE (K1) obtained from TEMPFILEITEMS are included into the E-R model of this class. Next it reads CDF B to find records having K1 as the first file. The first such record is a list containing K1, K2, L3. The next file in this list is BRANCH (K2) which has a BR type of "inv of_has_wards" with EMPLOYEE (K1). As K1 and K2 are kernel entities with the BR type of "inv_of_has_wards" the procedure applies rule **2** of Decision Table 2 to derive the following relationship name:

<EMPLOYEE belongs to BRANCH>

This relationship name together with the items of BRANCH obtained from TEMPFILEITEMS are then included into the E-R model of ABOUT EMPLOYEE.

The next file after K2 in the above CDF B record is L3. As L3 is a pure lookup entity the procedure ignores it and proceeds to read in the next record of CDF B having K1 as the first file. The record is a list containing K1, L1. However, since the next file in this list, namely RACE_CODE (L1) is a pure lookup entity, the procedure ignores it.

As there are no more records in CDF B having K1 as the first file, the procedure starts to read the CDF A file. The first record of CDF A is a list containing K1, S1. As S1 is a subtype entity with a BR type of "has_subtype" to K1, the procedure applies rule 1 of Decision Table 1 to derive the relationship name:

<EMPLOYEE is a MANAGER>

This relationship name together with the items of MANAGER obtained from TEMPFILEITEMS are then included in the E-R model of ABOUT EMPLOYEE class.

Next, the procedure reads the CDF B file to look for records containing S1 as the first file. However, there are no such records. It then proceeds to read in the next record of CDF A. This record contains K1, S2. Using the same rule as applied to S1 above, the procedure derives the following relationship name:

<EMPLOYEE is a NON-MANAGER>

This relationship name together with the items of NON-MANAGER obtained from TEMPFILEITEMS are then included in the E-R model of ABOUT EMPLOYEE class.

The procedure next reads the CDF B file to look for records containing S2 as the first file. However, there are no such records. It proceeds to read in the next record of CDF A. This record contains K1, A1, K3. As A1 (BILLINGS) is an associative entity and K3 (PROJECTS) is the other entity that associates A1, the procedure first applies rule **5***a* of Decision Table 1 using K1 and K3 to derive the following relationship name:

<EMPLOYEE has PROJECTS>

This relationship name together with the items of PROJECTS obtained from TEMPFILEITEMS are then included into the E-R model of ABOUT EMPLOYEE class. Next the procedure reads CDF B file to look for records with K3 as the first file. There is one such record, namely K3, L2. However, as L2 (CUSTOMER) is a pure lookup entity, the record is ignored. The procedure then applies rule **5***b* of Decision Table 1 on the current CDF A record, namely the list containing K1, A1, K3, to derive the following relationship name:

<EMPLOYEE and PROJECT have BILLINGS>

This relationship name together with items of BILLINGS obtained from TEMPFILEITEMS are then included in the E-R model of ABOUT EMPLOYEE Class.

The procedure next reads the CDF B file to look for records with A1 as the first file. However, there is no such file. It then proceeds to read in the next CDF A record. This record contains K1, C1. Applying rule 4 of Decision Table 1, the procedure derives the following relationship name

<EMPLOYEE has SKILLS)

This relationship name together with the items of SKILLS obtained from the TEMPFILEITEMS file are then included in the E-R model of ABOUT EMPLOYEE class.

The procedure next reads the CDF B file to look for records having C1 as the first file. However, there are none and so it proceeds to read the next CDF A record. However, there are no more CDF A records. This ends the E-R model translation for the ABOUT EMPLOYEE class from its class definition.

Besides the procedure just described, the E-R Model Translator also has another procedure which creates a class using CDF C to contain all the pure lookup entities. For the personnel system example, this procedure creates the following class:

ABOUT PURE LOOKUP ENTITIES
<RACE_CODE>
Race_Code
Race_Desc
<CUSTOMER>
Cust_No

Cust_Name
<COUNTRY>
Country_No
Country_Name

Extension To Existing Modules

Besides extending Semantics Extractor **12**, in one embodiment Model Purifier **26**, and/or Report Item Selector **16** are also extended. The extension of Model Purifier **26** provides an interface to allow a user to specify, for two opposite binary relationships of UU (unique key to unique key), which binary relationship is the "has_subtype" and which is the "inverse of has_subtype" type, if a file has only one sub-type. However, this is not necessary for embodiments using the SQL language because it is possible to know this from the primary and foreign keys.

The extension of Report Item Selector **16** provides an interface that displays all or selected classes and their E-R model, as well as allowing a user to formulate a query by picking the desired class attributes from the E-R model of the classes.

Model purifier **26** and Security Model Specifier **29** are optional modules to this embodiment.

In conclusion, this embodiment provides an alternative way of presenting the database items for end-users to select to formulate their queries. This presentation of using classes and their E-R models is more intuitive and meaningful than a single list of items. As a result, this embodiment makes it easier for end-users to formulate relational database queries.

INTEGRATION WITH ELECTRONIC MAIL

In one embodiment of this invention, the end-user query facility is integrated with an electronic mail (E-mail) system to allow a user located at a site with no on-line access to the application database but with connection to an E-mail system to still be able to make a query of the application database. The E-mail system provides the delivery mechanism for the user's query to be transmitted to the site where the query will be processed, and for the query results to be returned to the user after processing. The integration with the E-mail system also provides one convenient method for all queries made via the E-mail system to be logged into a log file. Information such as user id, items selected by user, the number of records extracted from database, the time of query, and the time taken to process the query can all be logged.

FIG. **19** shows one example of this embodiment. User A at a site with no on-line access to the Application Database **39** interfaces with Report Item Selector **16** of Information Scout **15** to select the items to be reported. Note that in this embodiment, the Information Scout consists only of the Report Item Selector which interfaces with the Send-Mail Agent instead of the Inference Engine **17**. The Send-Mail Agent **41** then takes the items selected and stores them into a file which is then attached to an E-mail. This E-mail is then sent to the Query Mailbox **40**. The Open-Mail Agent **42** periodically checks for E-mail in the Query Mailbox **40**. If there is an E-mail, it reads the E-mail to get the file attached. This file contains the items selected by the User A. It then passes the items selected to the Inference Engine. Note that in this embodiment the Inference Engine **17** interfaces with the Open-Mail Agent **42** instead of the Report Item Selector **16**. The Inference Engine then identifies one or more database files which contain the desired information and searches the Knowledge Base **14** to determine the linkages connecting the identified files. Program Generator **18** then generates a Source Program **19** based on the linkages inferred. Source Program **19** is then compiled and executed against the Application Database by the Compiler/Executor **20**. The query result obtained is then composed into an E-mail by the Open-Mail Agent **42** and sent to User A Mailbox **43**. User A obtains his query result by reading this E-mail.

The following describes in greater detail the Send-Mail Agent **41** and the Open-Mail Agent **42** based on the integration with Microsoft Mail, a product of Microsoft Corporation, USA.

The procedure for the Send-Mail Agent **41** is given as a representative section of code in Program Listing **1**. It is written in Microsoft's Visual Basic under Microsoft's Windows operating system and uses the Microsoft's Electronic Forms Designer as well as the Microsoft's Messaging Application Program Interface (MAPI) to interface with the Microsoft Mail E-Mail System.

The procedure works as follows. After User A has selected his items using the Report Item Selector **39** which has been implemented using Microsoft's Electronic Forms Designer, the Send-Mail Agent **41** is invoked. This Agent first stores the items selected into a temporary file called query.dat which is then assigned to another file called FName. Next a function called WriteMessage is called to attach this file to an E-Mail created by Report Item Selector **39**. The WriteMessage function is one of the functions provided by the Microsoft's Electronic Forms Designer. This function uses the command MEFAddAttachment ( ) to attach the FName file to the E-Mail. This E-Mail is then posted to the Query Mailbox **40** by using the command MAPISendMail. This completes the description of Send-Mail Agent **41**.

The procedure for Open-Mail Agent **42** is given as a representative section of code in Program Listing **2**. It is written in C language under Microsoft's Windows operating system and also uses Microsoft Mail's Messaging Application Program Interface (MAPI) to interface with Microsoft Mail E-Mail system.

This procedure works as follows. It periodically checks Query Mailbox **40** of the E-Mail system using the subroutine vTimerServeProc( ) running under subroutine WinProc( ). VTimerServeProc( ) in turn uses the MAPI command MAPIFindNext to find out whether there is any E-Mail in the mailbox. If there is, it then issues the MAPI command MAPIReadMail to read the E-Mail. At the same time it logs the time the message is read into the server log file (which is one of the two files under Log File **44**, the other being queue log file) by calling the sub-routine vServerLog. It next processes the attachment in the E-Mail. The attachment is a file where the items selected by a user as his query is stored. This file is called query.dat. vTimerServerProc next calls vServeAttach to process this attachment. vServeAttach first logs the user name, the subject and the date received into the server log file. It then calls Inference Engine **17** which is implemented as a sub-routine called lAppPHEng. This subroutine also contains Program Generator **18** to generate the Powerhouse Quiz source program based on the linkages determined by Inference Engine **17**. It next uses Compiler/Executor **20** to get the query result from the database. Compiler/Executor **20** is implemented as a sub-routine called lAppPHRet. The query result is stored in a file called RESULT.XXX where XXX can be XLS, MDB, TXT, DBF which are the file extensions for the different file formats available. Next the vServeAttach logs the number of rows of records retrieved from the database that make up the query result. It also logs the selection filter used in the user query as well as the items selected into the queue log file of Log File **44** using the sub-routine vQueueLog. Lastly, it composes an E-Mail to contain the query result and then use the MAPISendMail command to send this E-Mail to User A Mailbox **43**. This completes the description of the Open-Mail Agent.

Thus by integrating the query facility to an E-Mail system, a user who has no on-line access to his application database is still able to make his query using the E-Mail system. Another advantage of this embodiment is that all queries made can be logged for historical analysis. Furthermore, the embodiment allows queries to be pro-

cessed in batch mode during off-peak hours whenever there is excessive load on the database system during peak hours.

ALTERNATIVE WAY OF PRESENTING QUERY RESULTS TO HELP PREVENT MISINTERPRETATION

This alternative embodiment provides an alternative way in which query results can be presented to help prevent users from misinterpreting their query results. Query results can possibly be misinterpreted by users because of their complexity, which is in turn due to the queries being complex. A complex query is one which is made up of many basic queries with each basic query having its own distinct result. What a user gets when he makes a complex query is a report in which these distinct basic query results have been compounded. This compound report is not always easy for the user to understand or interpret. This embodiment breaks down a complex query into its basic components and processes each basic component (which is a basic query) separately so that what a user finally receives as a result is a disjoint set of distinct basic query results which the user can easily understand or interpret.

This alternative embodiment involves adding a new module called the Knowledge Thread Analyser **50** to the query facility of FIG. **14** described earlier. An exemplary main flowchart for this alternative embodiment is shown in FIG. **20**.

Knowledge Thread Analyser

Before we describe the function of the Knowledge Thread Analyser **50**, let us explain how query results can be misinterpreted by illustrating two different cases.

Case A:

Suppose we have two database files as follows:

File	Item	Key
INVOICE	Inv-No Inv-Amt Customer	Unique key is Inv-No
INVOICE_LINES	Inv-No Part-No Qty Price	Repeating key is Inv-No

Suppose also that INVOICE has only one record as follows:

Inv-No	Inv-Amt
1	$1000

and INVOICE_LINES has 3 records as follows:

Inv-No	Part-No	Qty	Price
1	A	2	$ 50
1	B	3	$100
1	C	7	$175

Suppose a user picks the items Inv-No, Inv-Amt, Part-No, Qty and Price as his query. The following query result would be produced after the query is processed.

Inv-No	Customer	Inv-Amt	Part-No	Qty	Price
1	IBM	$1000	A	2	$ 50
1	IBM	$1000	B	3	$100
1	IBM	$1000	C	7	$175

This result is correct but can be misinterpreted by a user because the Inv-Amt of $1,000 is repeated for every record of INVOICE_LINES. A user may total up the Inv-Amt to get what he thinks is the total invoice amount, namely $3000. This is wrong as there is only one invoice amount which is $1000.

This query is complex and can be broken down into two basic queries. One basic query is about finding out the amount for each invoice for each customer. The items Inv-No, Customer and Inv-Amt from the INVOICE file would provide this information. The other basic query is about finding out the details of each invoice, which details can be obtained from the items Inv-No, Customer, Part-No, Qty, and Price. The above query result is a compound report of these two basic queries. However, if both these basic queries were processed separately we would get a disjoint set of basic query results as follows:

Inv-No	Customer	Inv-Amt	Part-No	Qty	Price
1	IBM	$1000			
1	IBM		A	2	$ 50
1	IBM		B	3	$100
1	IBM		C	7	$175

Notice that with this disjoint set of basic query results the Inv-Amt is now correctly presented as a single value of $1000 and not a repeating value of $1000.

This case can be identified by the fact that the query involves two files in which they have a "one to many" relationship. Alternatively, we can say that their binary relationships is a UR (unique to repeating) or UN (unique to non-key). We can also say that their binary relationship is a "has_children" or "has_wards" type. Also, one of the items selected is from the file whose unique key is used in the binary relationship, this item being non-key and numeric. For the example above, INVOICE is the file whose unique key Inv-No is used in the binary relationship with INVOICE_LINES. Inv-Amt is one of the items selected as part of the query and it is non-key and numeric.

Consider another case which can also lead to misinterpretation by users.

Case B:

Suppose we have three database files as follows:

File	Item	Key
EMPLOYEE	Emp_No Emp_Name	Unique key is Emp_No
SKILLS	Emp_No Skill	Repeating key is Emp_No
PAYS	Emp_No Month Salary	Repeating key is Emp_No

Suppose also that these three files contain the following records:

EMPLOYEE:	Emp_No	Emp_Name	
	1	John	
	2	Sally	
SKILLS:	Emp_No	Skill	
	1	COBOL	
	1	Fortran	
	1	Ada	
	2	C++	
	2	COBOL	
PAYS:	Emp_No	Month	Salary
	1	Jan	1000

-continued

1	Feb	1000
2	Jan	2000
2	Feb	2000

Suppose a query is formulated by picking the items Emp_Name, Skill, Month, and Salary. Let us first consider the case of using PowerHouse to process this query, followed by the case of using SQL to process this query. The reason for doing this is that both PowerHouse and SQL give different results, but each may still lead to misinterpretation by users. The following result would be produced after this query is processed by PowerHouse.

Emp Name	Skill	Month	Salary
John	COBOL	Jan	1000
John	Fortran	Feb	1000
John	Ada		
Sally	C++	Jan	2000
Sally	COBOL	Feb	2000

A user may then use this result to find out the all Salary records of each employee with COBOL skill. As the first and fifth row of the query result contain COBOL, he would think that the Salary records come from these two rows only, namely:

Emp_Name	Month	Salary
John	Jan	1000
Sally	Feb	2000

However, this is not totally correct as the correct result should include all the Salary records of John and Sally both of whom have COBOL skill. The correct result is as follows:

Emp_Name	Month	Salary
John	Jan	1000
John	Feb	1000
Sally	Jan	2000
Sally	Feb	2000

Let us now consider the case of using SQL to process same query involving Emp_Name, Skill, Month and Salary. The following result would be produced by SQL:

Emp_Name	Skill	Month	Salary
John	COBOL	Jan	1000
John	COBOL	Feb	1000
John	Fortran	Jan	1000
John	Fortran	Feb	1000
John	Ada	Jan	1000
John	Ada	Feb	1000
Sally	C++	Jan	2000
Sally	C++	Feb	2000
Sally	COBOL	Jan	2000
Sally	COBOL	Feb	2000

This result could be misinterpreted by a user because the Salary of John and Sally is repeated many times. He may total up the Salary column to find the total Salary for John and Sally. What he would get respectively is 6000 and 8000. This is incorrect. The correct result is 2000 and 4000, respectively since John's salary is 1000 each in January and February and Sally's salary is 2000 each in January and February.

This query is complex and involves two multi-valued dependencies of EMPLOYEE, namely SKILLS and PAYS. It can be broken down into two basic queries. One basic query is about employees and their skills and the other basic query is about employees and their pays. The different query results by PowerHouse and SQL shown above are compound reports of these two basic queries. On the other hand the disjoint set of results for these two basic queries are:

Emp_Name	Skill	Month	Salary
John	COBOL		
John	Fortran		
John	Ada		
Sally	C++		
Sally	COBOL		
John		Jan	1000
John		Feb	1000
Sally		Jan	2000
Sally		Feb	2000

Notice now that with this report a user could easily find out correctly all salary records of employees who have COBOL skill. Also, with these two disjointed basic reports, the user would not incorrectly total up the salary of John and Sally.

This case can be identified by the fact that the query involves more than one multi-valued dependencies (MVD). For example, in the above case the attribute or item SKILL is multi-dependent on the attribute or item Emp_No, i.e. each employee has a well-defined set of skills. Similarly, the attributes or items Month and Salary are also multi-dependent on Emp_No. Alternatively, we can say that it involves at least three files in which one file has a "one-to-many" relationship or a UR (unique-to-repeating) or a UN (unique to non-key) binary relationship with each of the other files or in other words, one file has a "has_children" or a "has_wards" binary relationship with each of the other files.

Let us now describe Knowledge Thread Analyser **50**. It takes as its input the knowledge thread (as defined earlier) determined by Inference Engine **17** based on the items selected by the user from a single class. Since the items selected by a user are obtained from a single class, since every knowledge thread determined by Inference Engine **17** has as its thread-head the root kernel entity of the class from which the items are selected. Knowledge Thread Analyser **50** then analyses this knowledge thread, breaking it down first into simple knowledge threads to resolve the Case B type of complex query problems, and second for each simple knowledge thread derived breaking it down further into smaller simple knowledge threads to resolve the Case A type of complex query problems. The new knowledge threads produced as a result of this analysis are then passed to Program Generator **18** to generate the corresponding source programs which are then compiled and executed to produce the query results.

Before we describe in detail this exemplary procedure, let us recap what a knowledge thread is by giving an example, because it will help in understanding this procedure. Suppose a user makes a query on the ABOUT EMPLOYEE class described earlier by selecting the items Emp_Name, Salary, Branch_Name, Br_Tot_Expenses, Country_Name, Skill, Month and Amount. The knowledge thread determined by Inference Engine **17** is as follows:

EMPLOYEE → BRANCH (NU) → COUNTRY (NU)
→ SKILLS (UR)
→ BILLINGS (UR)

This knowledge thread has one thread-head but many thread-ends, with the thread-head being the root kernel entity of the class. What we want to do is to breakdown this complex thread into many simple threads, where each simple thread has one thread-head and only one thread-end. This complex knowledge thread can be broken down into three simple knowledge threads as follows:

EMPLOYEE→BRANCH (NU)→COUNTRY (NU)

EMPLOYEE→SKILLS (UR)

EMPLOYEE→BILLINGS (UR)

Two different types of simple threads can be derived from a complex thread determined by Inference Engine **17**. One type is the NU or RU threads in which all the binary relationships between two adjacent files in the thread is NU or RU. Another type is the UR threads in which all the binary relationships between two adjacent files is UR. One characteristic of the UR simple threads is that their thread-head is same as the thread-head of the complex thread. As for the NU or RU simple threads, their thread-head is the file on the complex thread which starts off the NU or RU relationship.

For example, the above complex knowledge thread has one NU simple thread, namely

EMPLOYEE→BRANCH (NU)→COUNTRY (NU)

and two UR simple threads, namely

EMPLOYEE→SKILLS (UR)

EMPLOYEE→BILLINGS (UR)

We shall now describe an exemplary embodiment of Knowledge Thread Analyser **50** in detail. It comprises the following steps:

Step 1: Analyze the knowledge thread determined by Inference Engine **17** and derive the NU or RU simple knowledge threads and the UR simple knowledge threads. Store the NU or RU simple knowledge threads in NU-RU Thread File and the UR simple knowledge threads in UR Thread File. This resolves the Case B type of complex query problem.

Step 2: Access the first simple knowledge thread in NU-RU Thread File. Starting from the thread-end analyze each pair of adjacent files for Case A type of complex query. If a pair has a Case A type of complex query, generate a new knowledge thread comprising files from the thread-end through to the first file in the pair of adjacent files being analyzed, this first file forming the thread-head of the new knowledge thread. Assign to this new knowledge thread its respective set of user selected items such that only its thread-head has user selected items which are both non-key and numeric. Store this new knowledge thread in New Thread File. Repeat this step for each of the remaining simple threads in NU-RU Thread File.

Step 3: Access the first simple knowledge thread in UR Thread File. Starting from the thread-head analyze each pair of adjacent files for Case A type of complex query. If a pair has a Case A type of complex query, generate a new knowledge thread comprising files from the thread-head through to the first file in the pair of adjacent files being analyzed, with the thread-head of the new knowledge thread being the same as the thread-head of the knowledge thread being analyzed. Assign to this new knowledge thread its respective set of user-selected items such that only its thread-end has user selected items which are both non-key and numeric. Store this new knowledge thread in New Thread File. Repeat this step for each of the remaining simple threads in UR Thread File.

Step 4: For each simple knowledge thread in UR Thread File combine it with those NU or RU simple threads in the NU-RU Thread File which have their thread-head matching a file in the UR simple knowledge thread. This complex thread becomes a new knowledge thread. Assign this new knowledge thread its respective set of user selected items such that only its UR thread-end has user-selected items which are both non-key and numeric. Store this new knowledge thread in New Thread File.

Step 5: Eliminate duplicate knowledge threads in the New Thread File.

Let us apply this procedure on the earlier example of a query on the ABOUT EMPLOYEE class to show how the above procedure works. When we apply step 1, we get the following NU simple knowledge threads

EMPLOYEE→BRANCH (NU)→COUNTRY (NU)

and the following UR simple knowledge threads:

EMPLOYEE→SKILLS (UR)

EMPLOYEE→BILLINGS (UR)

These simple threads are stored in the NU-RU Thread File and the UR Thread File, respectively. Next we use step 2 on the NU simple knowledge thread. The first pair of adjacent files starting from the thread-end are COUNTRY and BRANCH. Since the user-selected item from COUNTRY file is Country_Name which is not numeric, there is no Case A type of complex query in this pair of adjacent files. So no new knowledge thread is generated for this pair. The next pair comprises BRANCH and EMPLOYEE. Since one of the user-selected items from BRANCH is Br_Tot_Expenses which is both non-key and numeric, there is a Case A type of complex query in this pair. The following new knowledge thread is generated as a result:

BRANCH→COUNTRY (NU)

The user-selected items assigned to it are Country_Name, Branch_Name, and Br_Tot_Expenses. Notice that only the thread-end, namely BRANCH, has the item which is both non-key and numeric, namely Br_Tot_Expenses. This new Knowledge Thread is then stored in the New Thread File.

As there are no more NU simple knowledge thread, we next apply step 3 on the UR simple knowledge threads. The first UR simple knowledge thread is

EMPLOYEE→SKILLS (UR)

There is only pair of adjacent files in this thread. Since one of the user-selected items from EMPLOYEE is Salary which is both non-key and numeric, there is a Case A type of complex query in this pair. The following new knowledge thread is generated as a result:

EMPLOYEE

The user-selected items assigned to it are Emp_Name and Salary. This new knowledge thread is then stored in the New Thread File.

The next UR simple knowledge thread is

EMPLOYEE→BILLINGS (UR)

For this pair of files, the following new knowledge thread is also generated:

EMPLOYEE

with Emp_Name and Salary being assigned to it. It is then stored in the New Thread File.

Next we apply step 4. We combine the NU simple thread EMPLOYEE→BRANCH (NU)→COUNTRY (NU) with

the UR simple knowledge thread EMPLOYEE→SKILLS (UR) to derive the following new knowledge thread:

EMPLOYEE → BRANCH (NU) → COUNTRY (NU)
→ SKILLS (UR)

The user-selected items assigned to it are Emp_Name, ranch_Name, Country_Name and Skill. This new knowledge thread is then stored in the New Thread File.

Another new knowledge thread is also formed as follows:

EMPLOYEE → BRANCH (NU) → COUNTRY (NU)
→ BILLINGS (UR)

The user-selected items assigned to it are Emp_Name, Branch_Name, Month, and Amount. This new knowledge thread is then stored in the New Thred File. Notice that only the UR thread-end, namely BILLINGS, has the user-selected item which is both non-key and numeric, namely Amount.

Step 5 is applied next. In this step the duplicate thread, namely EMPLOYEE, in the New Thread File is eliminated. This completes the analysis of the complex knowledge thread determined by the Inference Engine **17** by the Knowledge Thread Analyser **50**.

In this embodiment Model Purifier **26** and Security Model Specifier **29** are optional modules.

Thus this embodiment ensures that end-users will not misinterpret their query results whenever they make complex queries as it ensures that the query results produced are presented as a disjointed set of basic query results.

Connections to Existing Database Applications

Many applications today are built using standard application program interface (API) that allow them to be easily linked to other software. One type of standard, which is a data access interface standard, relates to the linking of applications to different database management systems (DBMS). Open Database Connectivity (ODBC) from Microsoft Corporation is one of the popular standards for such a purpose. This type of standard allows applications to be developed, compiled and shipped without targeting a specific DBMS so long as they make function calls to a database using the API defined by this standard. In order to link such an application to the DBMS of their choice, users then add a module called a database connectivity driver specific to that DBMS. A database connectivity driver implements the function calls of the standard API according to the requirements of the DBMS they are designed for. Many of these applications provide a primitive ad-hoc query capability in that users of these applications must understand the database model in order to perform ad-hoc query. This invention relates to a novel approach for the construction of this end-user query facility through the use of what we called a Query Facility Connectivity Driver so that we seamlessly link the end-user query facility to these applications to enhance their query capability while still allowing the linking of these applications to the DBMS of the users' choice using a database connectivity driver specific to that DBMS. In accordance with the teachings of the invention this seamless linking of the application to the end-user query facility is achieved by adding the Query Facility Connectivity Driver to the application in the same way as adding a database connectivity driver to link the application to a DBMS. At the same time a user can still link his application to the DBMS of his choice as he can continue to add one or more database connectivity drivers specific to that DBMS in the same manner as before. All these are made possible by constructing the end-user query facility to include a Query Facility Connectivity Driver and by constructing this driver firstly to have the same standard data access interface as a database connectivity driver so that it can be added as a module to the application just like adding a database connectivity driver, and, secondly, to have an interface that is compliant to the same data access interface standard as the application so that the end-user query facility can be linked to the DBMS of the user's choice by adding a database connectivity driver specific to that DBMS in the same way as the application can be linked to the DBMS by adding a database connectivity driver specific to that DBMS.

The following describes one embodiment of this invention using ODBC from Microsoft Corporation as the standard data access interface. However, it is to be understood that the teachings of this invention are equally applicable to any other standard data access interface.

Before we describe this invention, let us first describe what ODBC is.

ODBC

In the traditional database world, an application is tied to a specific database management system (DBMS). An application could be a payroll system or a spreadsheet with query capability. Such applications are usually written using embedded SQL. Though embedded SQL is efficient and portable across different hardware and operating systems, the source code must be recompiled for each new environment. Also, it is not optimal if the applications need to access data in different DBMS such as IBM DB2 or Oracle. One version of the application would have to be recompiled with the IBM precompiler and another version with the Oracle precompiler resulting in the user having to purchase two products of the same application instead of one in order to be able to access both DBMS.

The ODBC interface defined by Microsoft Corporation, on the other hand, allows an application to be developed and compiled without targeting a specific DBMS. The users of such an application then add modules called database connectivity drivers to link the application to their choice of database management systems. These database drivers are dynamic link libraries that an application can invoke on demand to gain access to a particular data source through a particular communications method much like a printer driver running under Microsoft's Windows. ODBC provides the standard interface that allows data to be shuttled between the applications and the data sources.

The ODBC interface defines the following:

- A library of ODBC function calls that allow an application to connect to a DBMS, execute SQL statements and retrieve results.
- The SQL syntax used is based on the X/Open and SQL Access Group (SAG) SQL CAE Specifications (1992).
- A standard set of error codes.
- A standard way to connect and log on to a DBMS.
- A standard representation for data types.

The ODBC architecture has four components, namely

- Application: performs processing of its application functions and calls ODBC functions to submit SQL statements and retrieve results.
- Database Connectivity Manager: loads the database connectivity drivers on behalf of the application.
- Database Connectivity Driver: processes ODBC function calls, submits SQL requests to a specific data source and returns results to the application.
- Data Source: consists of the data the user wants to access, the associated operating system, the DBMS and the network platform (if any) used to access the DBMS.

FIG. **21** shows the relationship among the four components of the ODBC architecture. Each component carries out different functions as listed below.

Functions of Application

Connects to the Data Source by specifying the data source name

Processes one or more SQL statements as follows:

The Application places the SQL text string in a buffer

If the statement returns a result the application assigns a cursor name for the statement or allows the Database Connectivity Driver to do so.

The Application submits the statement for prepared or immediate execution.

If the statement creates a result, the Application can enquire about the attributes of the result such as the number of columns and the name and type of a specific column. It assigns buffers for each column in the result and fetches the result.

If the statement causes an error, the Application retrieves error information from the driver and takes appropriate action.

Ends each transaction by committing it or rolling it back.

Terminates the connection when it has finished interacting with the Data Source.

Functions of Database Connectivity Manager

A dynamic-link library (DLL) whose primary purpose is to load ODBC drivers.

Also processes several ODBC initialization and information calls.

Passes ODBC functions calls from Application to Database Connectivity Drivers.

Performs error and state checking.

Functions of Database Connectivity Drivers

A DLL that implements ODBC function calls and interacts with a Data Source.

It performs the following tasks in response to ODBC function calls from an Application:

Establishes a connection to the Data Source.

Submits requests to the Data Source.

Translates data to or from other formats.

Return results to the Application

Declare and manipulate cursors

Tasks performed by Data Source

Requires its name, a user ID and a password to be specified by a user of the Application for it to be connected.

Processes SQL requests received from the Database Connectivity Driver.

Returns result to the Database Connectivity Driver.

Query Facility Connectivity Driver

FIG. **22** shows Query Facility **65** linked to Application **60** using Query Facility Connectivity Driver **64**. Query Facility Connectivity Driver **64** "slots" in-between Database Connectivity Manager **61** and Database Connectivity Driver **62**. replacing Report Item Selector **16** of Information Scout **15** (FIG. **1**) used in previous embodiments of the query facility.

Query Facility Connectivity Driver **64** has the same or substantially similar ODBC interface as Database Connectivity Driver **62** so that it can be used by Application **60** as though it is a Database Connectivity Driver. However, unlike Database Connectivity Driver **62**, it implements many of the ODBC functions in a different way, since it is used to connect Application **60** to Query Facility **65** instead of to Application Database__. Table 3 gives a comparison of the different implementation of the OBDC calls between Database Connectivity Driver **62** and Query Facility Connectivity Driver **64**.

Let us illustrate with an example how an Application **60** connects to Query Facility **65** through Query Facility Connectivity Driver **64**, and uses it to make powerful ad-hoc queries easily. We shall illustrate using Microsoft Excel spreadsheet as Application **60** making a query on an Application Database serving as a personnel system having the following database tables:

Table	Columns	Key
EMPLOYEE	Emp_No	Primary Key is Emp_No
	Emp_Name	
	Sex	
	Address	
PROJECT	Proj_No	Primary Key is Proj_No
	Proj_Name	
	Proj_Manager	
BILLINGS	Emp_No	Primary Key is Emp_No & Proj_No
	Proj_No	Foreign Key is Emp_No referencing EMPLOYEE
	Month	Foreign Key is Proj_No referencing PROJECT
	Amount	

After semantic extraction of this database model by Semantics Extractor **12** of Query Facility **65**. Keyword Library **13** contains the following keywords:

Emp_No	Emp_Name	Sex	Address
Project_No	Proj_Name	Proj_Manager	
Pay_No	Month	Amount	

and Knowledge Base **14** contains the following knowledge threads:

EMPLOYEE→BILLINGS

PROJECT→BILLINGS

EMPLOYEE→BILLINGS→PROJECT

Suppose a user wants a list of all employee names and the names of the projects they are working on. We would now like to describe how the user uses Microsoft Excel to make this query. We would also like to describe the various actions taken by Query Facility Connectivity Driver **64** in response to the ODBC calls made by Microsoft Excel in performing this query. But before we do so let us explain how Database Connectivity Driver **62** and Query Facility Connectivity Driver **64** are configured so that they can be used by Application **60** which is Microsoft Excel in this query example.

Let us assume that a Database Connectivity Driver **62** specific to DBMS **63** is used for the personnel system Application Database which has been loaded in. The user first starts a program called the ODBC Administrator. A window for adding data sources is then displayed. He presses the "Add" button to add a data source. Another window is then displayed to show a list of database connectivity drivers that have been loaded in. He selects the required Database Connectivity Driver **62** for his query on the personnel system. This driver then displays a window asking him to define his data source. He defines his data source to be the Application Database containing the personnel system and gives it the name "PERSONNEL_SYSTEM". This completes the configuration of Database Connectivity Driver **62**.

Let us now assume that Query Facility Connectivity Driver **64** has also been loaded in. The user again starts the ODBC Administrator to display a window for adding data sources. He presses the "Add" button to add a data source. Another window showing a list of database connectivity drivers including Query Facility Connectivity Driver **64** appears. This time he selects Query Facility Connectivity Driver **64**. This driver then displays a window asking him to define his data source. He defines his data source to be Keyword Library **13** of Query Facility **65** and gives it the name "ABOUT_PERSONNEL". This Keyword Library **13** for the query example contains the keywords of the person-

nel system. At this point we have only partly configured Query Facility Connectivity Driver 64 because, unlike Database Connectivity Driver 62, we need to configure another data source. Another window is displayed for the user to configure another data source. He configures the "PERSONNEL_SYSTEM" as the other data source which by the way is also the data source of Database Connectivity Driver 62. This completes the configuration of Query Facility Connectivity Driver 64.

Let us now describe how the user perform his query on the personnel system using Microsoft Excel aided by Query Facility 65. The user first launches Microsoft Excel. He then starts up MS-Query which is a module within Microsoft Excel for performing ad-hoc query. A dialog box appears showing him a number of data sources that he can connect to. He selects "ABOUT_PERSONNEL" as his data source. (If he had used Microsoft Excel of the prior art without Query Facility 65 and Query Facility Connectivity Driver 64 as taught by this embodiment, the "ABOUT_PERSONNEL" data source would not be shown on the dialog box and he would have selected "PERSONNEL_SYSTEM" as his data source.) MS-Query then makes the following ODBC calls in the order given to connect to the selected data source:

SQLAllocEnv
SQLAllocConnect
SQLGetInfo
SQLDriverConnect

The SQLAllocEnv call causes Database Connectivity Manager 61 to allocate storage for information about the ODBC environment as well as for special information required to run Query Facility Connectivity Driver 64 while the SQLAllocConnect call causes Query Facility Connectivity Driver 64, firstly, to allocate storage for information about the connection to the "ABOUT_PERSONNEL" data source and, secondly, to make the same call to Database Connectivity Driver 62 to allocate storage for information about the connection to the "PERSONNEL_SYSTEM" data source. The SQLGetInfo call causes Query Facility Connectivity Driver 64 to return its version number. The SQLDriverConnect call causes Query Facility Connectivity Driver 64 to do a number of things. Firstly, it connects to the "ABOUT_PERSONNEL" data source which is Keyword Library 13 of Query Facility 65. Secondly, it prompts the user for the password to Keyword Library 13. Thirdly, it makes the same call to Database Connectivity Driver 62 so as to connect to the "PERSONNEL_SYSTEM" data source which is the Application Database containing the personnel system. (If Microsoft Excel of the prior art had been used without Query Facility 65 and Query Facility Connectivity Driver 64 as taught by this embodiment, these same four ODBC calls would result in different actions being taken. The first call would cause Database Connectivity Manager 61 to allocate storage for information about the ODBC environment while the second call would cause Database Connectivity Driver 62 to allocate storage for information about the connection to the "PERSONNEL_SYSTEM" data source. The third call would cause Database Connectivity Driver 62 to return its version number while the last call would cause Database Connectivity Driver 62 to connect to the "PERSONNEL_SYSTEM" data source.)

After the data sources have been connected, MS-Query issues the SQLGetFunctions call to find out what DBMS 63 functions are supported by Query Facility Connectivity Driver 64. Query Facility Connectivity Driver 64 first passes the call to Database Connectivity Driver 62. It then takes the return results from Database Connectivity Driver 62 and modifies them to delete those functions it does not support before returning the results to MS-Query. Two of the functions it does not support are the 'update' and 'create' functions, since Query Facility 65 is intended only for query. (If Microsoft Excel had been used without Query Facility 65 and Query Facility Connectivity Driver 64, this call would be made to Database Connectivity Driver 62 and the results returned would be all the functions supported by the DBMS including the 'create' and 'update' functions.)

Next MS-Query issues a number of SQLGetInfo calls. These calls cause Query Facility Connectivity Driver 64 to return information about itself and the data source it is connected to, which is "ABOUT_PERSONNEL". (If Microsoft Excel had been used without Query Facility 65 and Query Facility Connectivity Driver 64, these calls would cause Database Connectivity Driver 62 to return information about itself and the data source it is connected to which is "PERSONNEL_SYSTEM".)

MS-Query next issues a SQLTables call. This call causes Query Facility Connectivity Driver 64 to prepare to return information regarding the table in Keyword Library 13. There is only one table in Keyword Library 13. This table is used to hold all the keywords in Keyword Library 13. For the query example using the personnel system let us call this table PERSONNEL. Following this MS-Query makes a SQLBindCol call to associate buffers to hold the table definition of the PERSONNEL table. It next issues a SQLFetch call to fetch and copy the table definition into the buffers. After this it makes a series of calls starting with SQLGetTypeInfo followed by SQLBindCol, and SQLFetch to get information on the data types used by the Application Database. As such information is also contained in Keyword Library 13, Query Facility Connectivity Driver 64 obtains them from Keyword Library 13. (If Microsoft Excel had been used without Query Facility 65 and Query Facility Connectivity Driver 64, the SQLTables call would cause the Database Connectivity Driver 62 to prepare to return information regarding the tables in the Application Database. These tables for the query example are the tables of the personnel system, namely EMPLOYEE, PROJECT and BILLINGS. The next two calls, namely SQLBindCol and SQLFetch, would cause Database Connectivity Driver 62 to associate buffers and then fetch and copy the table definitions of the EMPLOYEE, PROJECT and BILLINGS tables into the buffers. The next series of calls starting with SQLGetTypeInfo followed by SQLBindCol and SQLFetch would cause Database Connectivity Driver 62 to obtain the information on data types used by the Application Database.)

MS-Query next issues a number of SQLGetInfo calls to get the keywords used by DBMS 63. Examples of keywords obtained are (i) whether a table is referred to as "table" or "file" by the specific DBMS and (ii) whether an owner is referred to as "owner" or "authorization id". These calls cause Query Facility Connectivity Driver 64 to in turn make the same calls to Database Connectivity Driver 62 to get the required information from DBMS 63. (If Microsoft Excel had been used without Query Facility 65 and Query Facility Connectivity Driver 64 these calls would cause Database Connectivity Driver 62 to return the required information direct to MS-Query.)

At this point MS-Query stops issuing more ODBC calls. Instead, using the information it has obtained from the ODBC calls it has made, it displays a dialogue box showing the name(s) of the table(s) from the selected data source. This data source is "ABOUT_PERSONNEL" which has only one table PERSONNEL and so only the table name PERSONNEL is shown. To formulate his query the user needs a list of the keywords of Keyword Library 13 to be displayed. He therefore requests through this dialog box that the all the columns of the "PERSONNEL" table be displayed. MS-Query then issues a series of calls comprising SQLColumns, SQLBindCol and SQLFetch. The first call asks Query Facility Connectivity Driver 64 to prepare to

return information about the columns of the PERSONNEL table in Keyword Library13. The second call causes Query Facility Connectivity Driver 64 to associate a result buffer with a column in the result set, the result set in this case being all the keywords in Keyword Library 13. The third call causes Query Facility Connectivity Driver 64 to fetch the keywords from Keyword Library 13 and place them in the result buffer. (If Microsoft Excel had been used without Query Facility 65 and Query Facility Connectivity Driver 64, the names of the tables displayed in the dialog box would have been EMPLOYEE, PROJECT and BILLINGS. If the user wanted only the columns of EMPLOYEE table and had made the appropriate selection, MS-Query would issue the same calls and these calls would have caused Database Connectivity Driver 62 to prepare to return information about the columns of the EMPLOYEE table, associate a result buffer and fetch the columns of EMPLOYEE from the Application database into the result buffer.)

After all the keywords of Keyword Library 13 have been fetched, MS-Query then displays them to the user as the columns of the PERSONNEL table. The user then formulates his query by selecting the appropriate columns from this table. Since his query is to get a list of employee names and the names of the projects they are working on, he selects the columns Emp_Name and Proj_Name. MS-Query then generates the following SQL statement:

Select 'PERSONNEL'.'Emp_Name', 'PERSONNEL'.'Proj_Name' From 'PERSONNEL' 'PERSONNEL'

It next makes a SQLExecDirect call to have this SQL statement executed. Query Facility Connectivity Driver 64 upon receiving this call parses the SQL statement to extract the items selected by the user which in this query example are Emp_Name and Proj_Name. It then calls Inference Engine 17 of Query Facility 65 to determine the access path through the personnel system using the items selected. Program Generator 18 is then called to use this access path to generate an SQL source program. For the query example the SQL source program generated is as follows:

Select 'EMPLOYEE'.'Emp_Name', 'PROJECT'.'Proj_Name'

From 'EMPLOYEE' 'EMPLOYEE', 'PROJECT' 'PROJECT', 'BILLINGS' 'BILLINGS'

Where 'EMPLOYEE'.'Emp_No'='BILLINGS'.'Emp_No', 'PROJECT'.'Proj_No'='BILLINGS'.'Proj_No'

Query Facility Connectivity Driver 64 then issues the SQLExecDirect call to Database Connectivity Driver 62 to have this SQL source program executed by DBMS 63 to obtain the desired information from the Application Database. (If Microsoft Excel had been used without Query Facility 65 and Query Facility Connectivity Driver 64, the user would have to understand the personnel system database model to formulate this query. This means that he would have to know firstly that it requires the use of the column Emp_Name from the EMPLOYEE table and the column Proj_Name from the PROJECT table. Secondly, since these two tables are not directly connected but instead are connected through the BILLINGS table he would have to know about the relationships between these tables. Therefore, in order to formulate this query he must therefore request that the columns of the three tables be displayed. Let us assume that this has been done and that the display shows the columns of the three tables with their appropriate joins. The user would then select the column Emp_Name of EMPLOYEE table and Proj_Name of PROJECT as his query. MS-Query would then generate the following SQL statement:

Select 'EMPLOYEE'.'Emp_Name', 'PROJECT'.'Proj_Name'

From 'EMPLOYEE' 'EMPLOYEE', 'PROJECT' 'PROJECT', 'BILLINGS' 'BILLINGS'

Where 'EMPLOYEE'.'Emp_No'='BILLINGS'.'Emp_No', 'PROJECT'.'Proj_No'='BILLINGS'.'Proj_No'

Next it would make the SQLExecDirect call to Database Connectivity Driver 62 to send this SQL statement to DBMS 63 to be executed to extract the desired information from the Application Database.)

MS-Query next issues a series of ODBC calls to get the results of the query to be displayed. It first makes the SQLNumResultCol call to find out how many columns of the query results will be retrieved. For the query example the number of columns to be retrieved is two, one being the Emp_Name and the other being the Proj_Name of the PERSONNEL table. It next issues the SQLDescribeCol and the SQLColAttributes calls to obtain information about these two columns. To get this information Query Facility Connectivity Driver 64 first maps these two columns of the PERSONNEL table to the actual database columns, namely the Emp_Name of EMPLOYEE table and the Proj_Name of PROJECT table. Next, it issues the same calls to Database Connectivity Driver 62 to get the required information from the Application Database. After this MS-Query issues the SQLBindCol call to ask Query Facility Connectivity Driver 64 to allocate buffers. Next it issues the SQLFetch call. The SQLFetch call causes Query Facility Connectivity Driver 64 to make the same call to Database Connectivity Driver 62 to fetch the query results from the Application Database and place them in the associated buffers. (If Microsoft Excel had been used without Query Facility 65 and Query Facility Connectivity Driver 64, MS-Query would issue the same SQLNumResultCol call to find out how many columns would be retrieved. As the columns selected are Emp_Name of EMPLOYEE and Proj_Name of PROJECT, the number of columns to be retrieved would be two. MS-Query would next issue the same two SQLDescribeCol and SQLColAttributes calls. This would cause Database Connectivity to directly obtain the required information about these columns from the Application Database. After this MS-Query would issue the SQLBindCol and SQLFetch calls which would cause Database Connectivity Driver 62 to associate buffers and to directly fetch the query results from the Application Database and place them in the associated buffers.)

At this point the user has his query results displayed by MS-Query. If he does not wish to make any more queries, he can quit from MS-Query. When he quits this causes MS-Query to issue the following calls:

SQLDisconnect

SQLFreeConnect

SQLFreeEnv

The first call causes Query Facility Connectivity Driver 64 to disconnect from Keyword Library 13 and to also make the same call to Database Connectivity Driver 62 to disconnect from the Application Database. The second call causes Query Facility Connectivity Driver 64 to free the connection handle to Keyword Library 13 and to make the same call to Database Connectivity Driver 62 to free the connection handle to the Application Database. The third call causes Query Facility Connectivity Driver 64 to free its environment handle and to make the same call to Database Connectivity Driver 62 to free its environment handle. (If Microsoft Excel had been used without Query Facility 65 and Query Facility Connectivity Driver 64, these three calls would cause Database Connectivity Driver 62 to disconnect from the Application Database, to free the connection handle to the Application Database and to free its environment handle.)

Note that in the above explanation for the sake of clarity a number of ODBC calls made repeatedly by MS-Query

have not been mentioned, e.g. SQLAllocStmt and SQLFreeStmt. The SQLAllocStmt call is used to cause Query Facility Connectivity Driver **64** to, firstly, allocate SQL statement data structures to contain whatever information about Keyword Library **13** or the driver itself that will be obtained subsequently through other ODBC calls and, secondly, to make the same call to Database Connectivity Driver **62** to allocate SQL statement handle to contain whatever information about the Application Database or the driver itself that will be obtained subsequently through other ODBC calls. The SQLFreeStmt call is used to get Query Facility Connectivity Driver **64** to free the allocated SQL statement data structures and to make the same call to Database Connectivity Driver **62** to free the allocated SQL statement handle. (If Microsoft Excel had been used without Query Facility **65** and Query Facility Connectivity Driver **64**, the first of these two calls would be used to cause Database Connectivity Driver **62** to allocate an SQL statement handle and the second would be used to cause Database Connectivity Driver **62** to free the allocated SQL statement handle.)

While this exemplary description of the Connectivity to Exisiting Database Applications embodiment has been described with reference to various embodiments described previously which utilize Keyword Library **13** as the data source of Query Facility Connectivity Driver **64** to generate a table for the user to select items for the query, it is to be appreciated that this embodiment is equally applicable for use with other embodiments described above in which E-R Model of Classes File **32** (e.g. FIG. **11**) is used as the data source, where each class in File **32** is treated as a separate table to conveniently allow the user to select items for the query.

A key benefit of this embodiment of the present invention is that it allows the many millions of existing applications used by many organizations such as Microsoft Excel spreadsheet, Microsoft Access DBMS, Lotus **123** spreadsheet, Powersoft PowerViewer or InfoMaker, Gupta Quest, Q+E Database Editor, etc. that are ODBC compliant or compliant to other data access interface standard to be reused for making powerful ad-hoc queries easily through seamless integration of Query Facility **65** thus saving on the cost of purchasing new tools to have more powerful ad-hoc queries as well as the cost of training users to use the new tools.

TABLE 3

Comparison of Implementation of ODBC Calls Between a Database Connectivity Driver and the Query Facility Connectivity Driver

ODBC API	Database Connectivity Driver	Query Facility Connectivity Driver
RETCODESQL_APISQLTables(HSTMT hstmt, UCHAR FAR *szTableQualifier, SWORD cbTableQualifier, UCHAR FAR *szTableOwner, SWORD cbTableOwner, UCHAR FAR *szTable Name, SWORD obTableName, UCHAR FAR *szTableType, SWORD cbTableType)	The driver prepares to return information regarding the tables in the Application Database.	The driver prepares to return information regarding the table which is used to hold all the keywords in the Keyword Library 13 of the Query Facility 65.
RETCODESQL_APISQLStatistics(HSTMT hstmt, UCHAR FAR *szTableQualifier, SWORD cbTableQualifier, UCHAR FAR *szTableOwner, SWORD cbTableOwner, UCHAR FAR *szTableName, SWORD cbTableName, UWORD fUnique, UWORD fAccuracy)	Sets up the driver to return statistics about the tables in the Application Database.	Sets up the driver to return statistics about the table which is used to hold all the keywords in the Keyword Library 13 of the Query Facility 65.
RETCODESQL_APISQLTablePrivileges(HSTMT hstmt, UCHAR FAR *szTable Qualifier, SWORD cbTableQualifier, UCHAR FAR *szTableOwner, SWORD cbTableOwner, UCHAR FAR *szTableName, SWORD cbTableName)	Sets up the driver to return the privileges of the Application Database's tables. Privileges may be "SELECT", "UPDATE", "DELETE", or "REFERENCES".	Sets up the driver to return the privileges of the Keyword Library's table which is used to hold the keywords. The only privilege allowed for the table is "SELECT" since Keyword Library 13 is used for read only.
RETCODESQL_APISQLColumnPrivileges(HSTMT hstmt, UCHAR FAR *szTableQualifier, SWORD cbTableQualifier, UCHAR FAR *szTableOwner, SWORD bTableOwner, UCHAR FAR *szTableName, SWORD cbTableName, UCHAR FAR *szColumnName, SWORD cbColumnName)	Sets up driver to return the privileges of the columns of a particular table in the Application Database. Privileges may be "SELECT", "UPDATE", "DELETE", or "REFERENCES".	Sets up the driver to return the privileges of the columns of the Keyword Library's table used to hold the keywords. The only privilege allowed for these columns "SELECT" since the Keyword Library 13 is used for read only.
RETCODESQL_APISQLSpecialColumns(HSTMT hstmt, UWORD fColType, UCHAR FAR *szTableQualifier, SWORD cbTableQualifier, UCHAR FAR *szTableOwner, SWORD cbTableOwner, UCHAR FAR *szTableName, SWORD cbTableName,	Sets up the driver to return the required columns.	Return SQL_SUCCESS, but on subsequent SQLFetch/SQLGetData, return SQL_NO_DATA_FOUND. There is no need to identify columns that uniquely identifies a row because the Query Facility 65 performs all table linking using its Inference Engine 17 without any intervention from the Application 60 or the user of the Application 60. There is no need to return information regarding

TABLE 3-continued

Comparison of Implementation of ODBC Calls Between a Database Connectivity Driver and the Query Facility Connectivity Driver

ODBC API	Database Connectivity Driver	Query Facility Connectivity Driver
UWORD fScope, UWORD fNullable)		columns that are automatically updated since the Query Facility 65 does not support updates to the Application Database or to the Keyword Library.
RETCODESQL_APISQLPrimaryKeys(HSTMT hstmt, UCHAR Far *szTableQualifier, SWORD cbTableQualifier, UCHAR FAR *szTableOwner, SWORD cbTableOwner, UCHAR FAR *szTableName, SWORD cbTableName)	Sets up the driver to return the required primary keys of the Application Database.	Return SQL_SUCCESS, and the return SQL_NO_DATA_FOUND at SQLFetch/SQLGetData.
RETCODE SQL_APISQLForeignKeys(HSTMT hstmt, UCHAR FAR *szPkTableQualifier, SWORD cbPkTableQualifier, UCHAR FAR *szPkTableOwner, SWORD cbPkTableOwner, UCHAR FAR *szPkTableName, SWORD cbPkTableName, UCHAR FAR *szFkTableQualifier, SWORD cbFkTableQualifier, UCHAR FAR * szFkTableOwner, SWORD cbFkTableOwner, UCHAR FAR *szFkTableName, SWORD cbFkTableName)	Sets up the driver to return the required foreign keys of the Application Database.	Return SQL_SUCCESS, and the return SQL_NO_DATA_FOUND at SQLFetch/SQLGetData.
RETCODESQL_APISQLAllocEnv(HENV FAR *phenv)	Allocate storage for information about the ODBC environment.	Allocate storage for information about the ODBC environment as well as special information required to run the Query Facility 65.
RETCODESQL_APISQLAllocConnect(HENV henv, HDBC FAR *phdbc)	Allocate storage for information about a connection to the Application Database as a data source.	Allocate storage for information about a connection to the Keyword Library 13 of Query Facility 65 as a data source. Also make same call to Database Connectivity Driver to allocate storage for information about a connection to the Application Database as a data source.
RETCODESQL_APISQLConnect(HDBC hdbcParam, UCHAR FAR *szDSN, SWORD cbDSN, UCHAR FAR *szUID, SWORD cbUID, UCHAR FAR *szAuthStr, SWORD cbAuthStr)	Connects to the Application Database.	Connects to the Keyword Library 13. Prompts the user for Keyword Library 13 password. Makes SQLAllocEnv and SQLAllocConnect calls to the Database Connectivity Driver. Finally, connects to the Application Database by making the same SQLConnect call to the Database Connectivity Driver.
RETCODESQL_APIDriverConnect(HDBC hdbc, HWND hwnd, UCHAR FAR *szConnStrIn, SWORD cbConnStrIn UCHAR FAR *szConnStrOut, SWORD cbConnStrOutMax, SWORD FAR *szpcbConnStrOut, UWORD fDriverCompletion)	Connects to the Application Database, or alternatively, prompt the user for more information.	Connects to the Keyword Library 13 of Query Facility 65. Prompts the user for Keyword Library 13 password. Finally, connects to the Application Database by making the same call to the Database Connectivity Driver.
RETCODESQL_APISQLBrowseConnect(HDBC hdbc, UCHAR FAR *szConnStrIn, SWORD cbConnStrIn, UCHAR FAR *szConnStrOut, SWORD cbConnStrOutMax, SWORD FAR *pcbConnStrOut)	Iteratively prompts for Application Database connection string.	Iteratively prompts for Keyword Library 13 connection string.
RETCODESQL_APISQLDisconnect(HDBC hdbcParam)	Disconnects from the Application Database.	Disconnects from the Keyword Library 13 and make same call to Database Connectivity Driver to disconnect from database.
RETCODESQL_APISQLFreeConnect(HDBC hdbcParam)	Free connection handle of Application Database.	Free Keyword Library 13 connection handle, and makes same call to Database Connectivity Driver to free database connection handle.
RETCODESQL_APISQLFreeEnv(HENV henv)	Frees the driver's environment handle.	Frees the driver's environment handle, and then makes same call to Database Connectivity Driver to free its environment handle.
RETCODESQL_APISQLExecute(HSTMT hstmt)	Executes the SQL statement associated with hstmt.	Executes the SQL statement associated with hstmt
RETCODESQL_APISQLExecDirect(HSTMT hstmt, UCHAR FAR *szSqlStr, SDWORD cbSqlStr)	Executes the SQL statement passed in the parameter szSqlStr.	Parses the SQL statement to extract the items or columns selected by the user and then calls the Inference Engine 17 of the Query Facility 65 to determine the access path through the Application Database using the selected items. After the Program Generator 18 has generated an

TABLE 3-continued

Comparison of Implementation of ODBC Calls Between a Database Connectivity Driver and the Query Facility Connectivity Driver

ODBC API	Database Connectivity Driver	Query Facility Connectivity Driver
		SQL source program for the access path, the driver then makes a SQLExecDirect call to the Database Connectivity Driver to execute the SQL source program and to return the query results to the user.
RETCODESQL_APISQLNativeSql(HDBC hdbcParam, UCHAR FAR *szSqlStrin, SDWORD cbSqlStrIn, UCHAR FAR *szSqlStr, SDWORD cbSqlStrMax, SDWORD FAR *pcbSqlStr)	Translates the SQL statement to the SQL dialect of the DBMS 63 used by the Application Database.	Parses the SQL statement it receives to extract the items or columns selected by the user and then calls the Inference Engine 17 of the Query Facility 65 to determine the access path through the Application Database using the selected items. After the Program Generator 18 has generated the SQL source program for the access path, the driver translates this SQL source program to the SQL dialect of the DBMS 63 used by the Application Database.
RETCODESQL_APISQLParamData(HSTMT hstmt, PTR FAR *prgbValue)	Functions to support parameters.	Pass call directly to Database Connectivity Driver.
RETCODESQL_APISQLPutData(HSTMT hstmt, PTR rgbValue, SDWORD cbValue)	Functions to support parameters.	Pass call directly to database Connectivity driver.
RETCODESQL_APISQLSetParam(HSTMT hstmt, UWORD jpar, SWORD fCType, SWORD fSqlType, UDWORD cbColDef, SWORD ibScale, PTR rgbValue, SDWORD FAR *pcbValue)	Functions to support parameters.	Pass call directly to Database Connectivity Driver.
RETCODESQL_APISQLDescribeParam(HSTMT hstmt, UWORD ipar, SWORD FAR *pfSqlType, UDWORD FAR *pcbColDef, SWORD FAR *pibScale, SWORD FAR *pfNullable)	Functions to support parameters.	Pass call directly to Database Connectivity Driver.
RETCODESQL_APISQLParamOptions(HSTMT hstmt, UDWORD crow, UDWORD FAR *pirow)	Functions to support parameters.	Passcall directly to Database Connectivity Driver.
RETCODESQL_APISQLNumParams(HSTMT hstmt, SWORD FAR *pcpar)	Functions to support parameters.	Pass call directly to Database Connectivity Driver.
RETCODESQL_APISQLGetInfo(HDBC hdbcParam, UWORD finfoType, PTR rgbInfoValue, SWORD cbInfoValueMax, SWORD FAR *pcbInfoValue)	Return required information, either about itself or about the Application Database.	Return required information, either about itself, about the Database Connectivity Driver, or about the Application Database.
RECTCODE SQL_APISQLGetTypeInfo(HSTMT hstmt, SWORD fSqlType)	Retrieve information regarding datatype supported by Application Database and driver	Pass call to Database Connectivity Driver.
RETCODESQL_APISQLGetFunctions(HDBC hdbcParam, UWORD fFunction, UWORD FAR *pfExists)	Retrieve information regarding functions supported in the drievr	Pass call to Database Connectivity Driver. But return result is modified to reflect functions that may not apply to the Query Facility e.g. Query Facility does not support create or update functions.
RETCODESQL_APISQLSetConnectOption(HDBC hdbcParam, UWORD fOption, UDWORD vParam)	Set connection options.	Pass call to Database Connectivity Driver.
RETCODESQL_APISQLSetStmtOption(HSTMT hstmt, UWORD fOption, UDWORD vParam)	Set statement options.	Pass call to Database Connectivity Driver.
RETCODESQL_APISQLGetConnectOption(HDBC hdbcParam, UWORD fOption, PTR pvParam)	Get connection options.	Pass call to Database Connectivity Driver.
RETCODESQL_APISQLGetStmtOption(HSTMT hstmt, UWORD fOption, PTR pvParam)	Get statement options.	Pass call to Database Connectivity Driver.
RETCODESQL_APISQLAllocStmt(	Allocate SQL statement handle.	Allocate SQL statement data structures and call

TABLE 3-continued

Comparison of Implementation of ODBC Calls Between a Database Connectivity Driver and the Query Facility Connectivity Driver

ODBC API	Database Connectivity Driver	Query Facility Connectivity Driver
HDBC hdbcParam, HSTMT FAR *phstmt)		SQLAllocStmt in the Database Connectivity Driver.
RETCODESQL_APISQLFreeStmt(HSTMT hstmt, UWORD fOption)	Free SQL statement handle	Free SQL statement data structures and make same call to Database Connectivity Driver to free its SQL statement handle
RETCODESQL_APISQLPrepare(HSTMT hstmt, UCHAR FAR *szSqlStr, SDWORD cbSqlStr)	Prepare an SQL statement for execution	Parses the SQL statement to extract the items or columns selected by the user and then calls the Inference Engine 17 to determine the access path through the Application Database using the selected items. After the Program Generator 18 has generated an SQL source program for the access path driver then makes a SQLPrepare call to the Database Connectivity Driver to prepare the SQL source program.
RETCODESQL_APISQLCancel(HSTMT hstmt)	Cancels execution of a SQL statement or command.	Pass call to the Database Connectivity Driver.
RETCODESQL_APISQLSetScrollOptions(HSTMT hstmt, UWORD fConcurrency, SDWORD crowKeyset, UWORD crowRowset)	Obselete	Obselete
RETCODESQL_APISQLSetCursorName(HSTMT hstmt, UCHAR FAR *szCursor, SWORD cbCursor)	Associate a cursor name with a hstmt.	Associate a cursor name with a hstmt. If the hstmt is also associated with the Database Connectivity Driver(eg, in the case of SQLExecute), then call the equivalent function in the Database Connectivity Driver.
RETCODESQL_APISQLGetCursorName(HSTMT hstmt, UCHAR FAR *szCursor, SWORD cbCursorMax, SWORD FAR *pcbCursor)	Return the cursor name for a statement handle	Return the cursor name for a statement handle If the hstmt is also associated with the Database Connectivity Driver(eg, in the case of SQLExecute), then call the equivalent function in the Database Connectivity Driver.
RETCODESQL_APISQLNumResultCols(HSTMT hstmt, SWORD DAR *pccol)	Returns the number of columns in the result set.	Returns the number of columns in the result set. If the hstmt is associated with the Database Connectivity Driver, then make the same call to the Database Connectivity Driver.
RETCODESQL_APISQLDescribeCol(HSTMT hstmt, UWORD icol, UCHAR FAR *szColName, SWORD cbColNameMax, SWORD FAR *pcbColName, SWORD FAR *(pfSqlType, UDWORD FAR *pcbColDef, SWORD FAR *pibScale, SWORD FAR *pfNullable)	Return information about the column of a table in the Application Database the user wants information about.	Return information about the selected keyword from the Keyword Library 13 the user wants information about. This keyword is first mapped to an actual database column of the Application Database by the driver which then issues the same call to the database Connectivity Driver 62 to get information about this database column. After it gets this information, the driver returns this information.
RETCODESQL_APISQLAttributes(HSTMT hstmt, UWORD icol, UWORD fDescType, PTR rgbDesc, SWORD cbDescMax, SWORD FAR *pcbDesc, SDWORD FAR *pfDesc)	Returns descriptor information about the database column of a table in the Application Database the user wants information about.	Returns descriptor information about the selected keyword from the Keyword Library 13 the user wants information about. This keyword is first mapped to an actual database column of the Application Database by the driver which then issues the same call to the Database Connectivity Driver 62 to get the descriptor information about this database column. After it gets this descriptor information, the driver returns this information.
RETCODESQL_APISQLBindCol(HSTMT hstmt, UWORD icol, SWORD fCType, PTR rgbValue, SDWORD cbValueMax, SDWORD FAR*pcbValue)	Associate a result buffer with a column in the result set.	Associate a result with a column in the result set. If the hstmt is assocaited with the database driver, then make the same function call to the Database Connectivity Driver.
RETCODESQL_APISQLFetch(HSTMT hstmt	Returns data for bound columns in the current row, and advances the cursor.	Returns data for bound columns in the current row, and advance the cursor. If the hstmt is associated with the Database Connectivity Driver, then call the equivalent function in the Database Connectivity Driver.
RETCODESQL_APISQLGetData(HSTMT hstmt, UWORD icol, SWORD fCType, PTR rgbValue, SDWORD cbValueMax, SDWORD FAR *pcbValue)	Returns result data for a single column in the current row.	Returns result data for a single column in the current row. If the hstmt is associated with the Database Connectivity Driver, then call the equivalent function in the Database Connectivity Driver.

TABLE 3-continued

Comparison of Implementation of ODBC Calls Between a Database Connectivity Driver and the Query Facility Connectivity Driver

ODBC API	Database Connectivity Driver	Query Facility Connectivity Driver
RETCODESQL_APISQLMoreResults(HSTMT hstmt)	Checks if more information is available for hstmt.	Checks if more information is available for hstmt. If the hstmt is associated with the Database Connectivity Driver, then call the equivalent function in the Database Connectivity Driver1.
RETCODESQL_APISQLRowCount(HSTMT hstmt, SDWORD FAR *pcrow)	Returns the number of rows associated with the Application Database attached to "hstmt".	Returns the number ofrows attached to "hstmt" Hstmt may be associated with the Application Database or the Keyword Library 13. If the hstmt is associsted with the Databse Connectivity Driver, then call the equivalent function in the Database Connectivity Driver.
RETCODESQL_APISQLSetPos(HSTMT hstmt, UWORD irow, UWORD fOption, UWORD fLock)	Sets position of cursor.	Sets position of cursor. If the hstmt is associated with the Database Connectivity Driver, then call the equivalent function in the Database Connectivity Driver.
RETCODESQL_APISQLExtendedFetch(HSTMT hstmt, UWORD fFetchType, SDWORD irow, UDWORD FAR *pcrow, UWORD FAR *rgfRowStatus)	Fetches data.	Not implemented. ODBCDLL provides a default implementation.
RETCODESQL_APISQLError(HENV henv, HDBC hdbcParam, HSTMT hstmt, UCHAR FAR *szSqlState, SDWORD FAR *pfNativeError, UCHAR FAR *szErrorMsg, SWORD cbErrorMsgMax, SWORD FAR *pcbErrorMsg)	Returns the most recent error.	Retrns the most recent error. If the henv, hdbcParam, or hstmt is associated with the Database Connectivity Driver, then call the equivalent function in the Database Connectivity Driver.
RETCODESQL_APISQLTransact(HENV henv, HDBC hdbcParam, UWORD fType)	Performs commit or rollback	Does nothing since commit/rollback does not apply to Query Facility 65.

What is claimed is:

1. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of linkages of the database model, each said linkage representing a relation between two of said database files in which a first file has a key that references an equivalent key of a second file;

a semantics extractor for reading said database model and extracting the semantics of said database model, and which stores in said knowledge base said set of linkages;

an application, which supports end-user query, to obtain from a user a designation of the information to be extracted from said database;

a query facility connectivity driver for connecting said application to said query facility, and for connecting said query facility to said database through a database connectivity driver;

an inference engine which, based upon said designation of information to be extracted from said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files; and

a program generator which accesses the linkages obtained by said inference engine and generates a program to extract said desired information from said database.

2. An end-user query facility as in claim 1 wherein said inference engine comprises:

means for inferring new acquired knowledge threads and storing said new acquired knowledge threads in said knowledge base;

means for determining an access path to said database using either said basic knowledge threads or said acquired knowledge threads as required to meet the user query.

3. An end-user query facility as in claim 2 wherein said acquired knowledge thread is derived through a combination in parallel of two or more of said basic knowledge threads such that one of the said basic knowledge threads has one or more of its consecutive files in common with the corresponding number of consecutive files starting from the thread head of another said basic knowledge threads.

4. An end-user query facility as in claim 3 wherein said thread head is a first file on a said basic knowledge thread.

5. An end-user query facility as in claim 1 wherein said semantics extractor further comprises means for reading source code of application programs that access said database, extracting the semantics of said application programs and storing in said knowledge base said set of linkages.

6. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of classes and a set of linkages of the database model, each said class represents a hierarchical grouping of a subset of said database files, each said linkage representing a relation between two of said database files in which a first file has a key that references an equivalent key of a second file;

a class generator for reading said database model and generating said set of classes and said set of linkages of the database model and which stores in said knowledge base said set of classes and said set of linkages;

an application, which supports end-user query, to obtain from an user choices based on said classes as a designation of the information to be extracted from said database;

a query facility connectivity driver for connecting said application to said query facility, and for connecting said query facility to said database through a database connectivity driver;

an inference engine which, based upon said designation of information to be extracted from said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files; and

a program generator which accesses the linkages obtained by said inference engine and generates a program to extract said desired information from said database.

7. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of classes and a set of linkages of the database model; each said class represents a hierarchical grouping of a subset of said database files, each said linkage representing a relation between two of said database files in which a first file has a key that references an equivalent key of a second file;

a class generator for reading said database model and generating said set of classes and said set of linkages of the database model and which stores in said knowledge base said set of classes and said set of linkages;

an application, which supports end-user query, to obtain from a user a designation of the information to be extracted from said database;

a query facility connectivity driver for connecting said application to said query facility, and for connecting said query facility to said database through a database connectivity driver;

an inference engine which, based upon said designation of information to be extracted from said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files;

a knowledge thread analyzer receiving as its input said linkage(s) determined by said inference engine, and which breaks down said linkage(s) into simple linkage (s); and

a program generator which accesses said simple linkages obtained by said knowledge analyzer and generates programs, one for each said simple linkage, to extract said desired information from said database as a plurality of simple query results, one for each generated program.

8. An end-user query facility as in claim 7 wherein a said simple query result is one that does not contain repeated numeric values in any of the column(s) that corresponds to a non-key numeric item of said database.

9. An end-user query facility as in claim 7 wherein a said simple query result is one that does not contain two or more columns belonging to items whose database files are multi-valued dependencies of another database file in said database.

10. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of linkages of the database model, each said linkage representing a relation between two of said database files in which a first file has a key that references an equivalent key of a second file;

a semantics extractor for reading said database model and extracting the semantics of said database model, and which stores in said knowledge base said set of linkages;

a keyword library which stores a set of keywords of said database model;

an application, which supports end-user query, to obtain from a user a designation of the information to be extracted from said database using said keyword library;

a first connectivity driver for connecting said application to said keyword library;

an inference engine which, based upon said designation of information to be extracted from said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files;

a first e-mail agent for said first connectivity driver to interface with an e-mail system, said first e-mail agent being used to post said designation of information to be extracted from said database to the mailbox of said inference engine;

a second e-mail agent for the said inference engine to interface with said e-mail system, the said second e-mail agent being used to access said mailbox of said inference engine to obtain said designation of information to be extracted from said database;

a program generator which accesses the linkages obtained by said inference engine and generates a program to extract said desired information from said database.

11. An end-user query facility as in claims 1, 6 or 7 wherein both said application and said query facility connectivity driver have the same data access interface that conforms to a data access interface standard.

12. An end user query facility as in claims 1, 6 or 7 wherein both said query facility connectivity driver and said database connectivity driver have the same data access interface that conforms to a data access interface standard.

13. An end user query facility as in claims 11 or 12 wherein said data access interface standard is Microsoft Corporation's Open Database Connectivity (ODBC) standard.

14. An end user query facility as in claim 11 or 12 wherein said data access interface standard is Apple Computer's Data Access Language (DAL) standard.

15. An end user query facility as in claims 1 or 10 wherein said semantics extractor comprises means for deriving basic knowledge threads comprising a set of linkages of said database model and storing said knowledge threads in said knowledge base.

16. An end-user query facility as in claim 15 wherein each said basic knowledge thread comprises a set of two or more of said database files serially linked together such that one file is serially linked to the next file through an item that has the same domain as the item in the next file and that this same item is a unique or repeating key of the next file.

17. An end-user query facility as in claim 15 wherein each said basic knowledge thread comprises a set of two or more

of said database files serially linked together such that one file has a repeating or foreign key that references the unique or primary key of the next file.

18. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of classes and a set of linkages of the database model, each said class represents a hierarchical grouping of a subset of said database files, each said linkage representing a relation between two of said database files in which a first file has a key that references an equivalent key of a second file;

a class generator for reading said database model and generating said set of classes and said set of linkages of the database model and which stores in said knowledge base said set of classes and said set of linkages;

a keyword library which stores a set of keywords of said set of classes including the names of said classes and their attributes;

an application, which supports end-user query, to obtain from a user a designation of the information to be extracted from said database using said keyword library;

a first connectivity driver for connecting said application to said keyword library;

an inference engine which, based upon said designation of information to be extracted form said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files;

a first e-mail agent for said first connectivity driver to interface with an e-mail system, said first e-mail agent being used to post said designation of information to be extracted from said database to the mailbox of said inference engine;

a second e-mail agent for the said inference engine to interface with said e-mail system, the said second e-mail agent being used to access said mailbox of said inference engine to obtain said designation of information to be extracted from said database; and

a program generator which accesses the linkages obtained by said inference engine and generates a program to extract said desired information from said database.

19. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of classes and a set of linkages of the database model, each said class represents a hierarchical grouping of a subset of said database files, each said linkage representing a relation between two of said database files in which a first file has a key that references an equivalent key of a second file;

a class generator for reading said database model and generating said set of classes and said set of linkages of the database model and which stores in said knowledge base said set of classes and said set of linkages;

a keyword library which stores a set of keywords of said set of classes including the names of said classes and their attributes;

an application, which supports end-user query, to obtain from a user a designation of the information to be extracted from said database using said keyword library;

a first connectivity driver for connecting said application to said keyword library;

an inference engine which, based upon said designation of information to be extracted form said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files;

a first e-mail agent for said first connectivity driver to interface with an e-mail system, said first e-mail agent being used to post said designation of information to be extracted from said database to the mailbox of said inference engine;

a second e-mail agent for the said inference engine to interface with said e-mail system, the said second e-mail agent being used to access said mailbox of said inference engine to obtain said designation of information to be extracted from said database;

a knowledge thread analyzer receiving as its input said linkage(s) determined by said inference engine, and which breaks down said linkage(s) into simple linkage (s); and

a program generator which accesses said simple linkages obtained by said knowledge analyzer and generates programs, one for each said simple linkage, to extract said desired information from said database as a plurality of simple query results, one for each generated program.

20. An end user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said application comprises:

means for obtaining from a user a user supplied key word indicative of the data desired to be extracted from said database;

means for obtaining from a user a user supplied key word indicative of the data desired to be extracted from said database;

means for determining all keywords in said database model having a predefined relationship with said user supplied keyword; and

means for causing a user to select one or more of said keywords having a predefined relationship, said one or more keywords thus selected serving as said designation of the information to be extracted from said database.

21. An end user query facility as in claims 6, 7, 18 or 19 wherein each said class comprises a data model of a subset of database files of said database model, the said data model comprising a tree.

22. An end user query facility as in claim 21 wherein said tree of said class has a root comprising a database file of kernel entity type and a plurality of branches each comprising a linkage of one or more database files from said subset of database files.

23. An end-user query facility as in claim 22 wherein said class generator comprises:

a semantics extractor to derive a set of binary relationships and to derive the entity type of each database file of said database model;

a means to derive said set of classes using said set of binary relationships and said entity type of each database file of said database model.

24. An end-user query facility as in claim 23 wherein said entity type is selected from a set of entity types which comprise a kernel entity type, a subtype entity type, a characteristic entity type, an associative entity type or a pure lookup entity type.

25. An end-user query facility as in claim 23 wherein said binary relationship comprises a linkage between two files in said database model in which one file has a repeating or foreign key that references the unique or primary key of the other file.

26. An end user query facility as in claim 25 wherein said set of binary relationships comprises "has_children", "has_wards", "inverse_of_pure_lookup" or "has_subtype" binary relationships.

27. An end user query facility as in claim 26 wherein said set of binary relationships further comprises "inverse_of_has_children", "inverse_of_has_wards", "pure_lookup" or "inverse_of_has_subtype" binary relationships.

28. An end user query facility as in claim 27 wherein said "inverse_of_has_children" binary relationship is one in which the target file is not a pure_lookup entity type and has a one-to-many relationship with the source file, and the source file does not have its own independent unique or primary key.

29. An end user query facility as in claim 27 herein said "inverse_of_has_wards" binary relationship is one which the target file is not a pure lookup entity type and has a one-to-many relationship with the source file, and the source file has its own independent unique or primary key.

30. An end user query facility as in claim 27 wherein said "inverse_of_has_subtype" binary relationship is one in which its target file has a one-to-one relationship with its source file and the source file is a subtype entity of the target file.

31. An end-user query facility as in claim 27 wherein said "pure_lookup" binary relationship is a binary relationship whose target file is a pure lookup entity.

32. An end user query facility as in claim 26 wherein said "has_children" binary relationship is one in which the source file is not a pure lookup entity type and has a one-to-many relationship with the target file, and the target file does not have its own independent unique or primary key.

33. An end user query facility as in claim 26 wherein said "has_wards" binary relationship is one in which its source file is not a pure lookup entity type and has a one-to-many relationship with its target file, and its target file has its own independent unique key or primary key.

34. An end user query facility as in claim 26 wherein said "has_subtype" binary relationship is one which its source file has a one-to-one relationship with its target file and the target file is a subtype entity of the source file.

35. An end-user query facility as in claim 26 wherein said "inverse_of_pure_lookup" binary relationship is a binary relationship whose source file is a pure_lookup entity.

36. An end-user query facility as in claim 23 wherein said semantics extractor further comprises means for deriving basic knowledge threads comprising a set of linkages of said database model and storing said basic knowledge threads in said knowledge base.

37. An end-user query facility extractor as in claim 36 wherein each said basic knowledge thread comprises a set of two or more of said database files serially linked together such that one file is linked to the next file through an item that has the same domain as the item in the next file and that this same item is a unique or repeating key of the next file.

38. An end-user query facility extractor as in claims 37 wherein each said basic knowledge thread comprises a set of two or more of said database files serially linked together such that one file has a repeating or foreign key that references the unique or primary key of the next file.

39. An end-user query facility ads in claim 22 wherein said class further comprise a set of relationship names each of which specifies the nature of the relationship between two adjacent database files in each branch of said tree of said class.

40. An end-user query facility as in claim 39 wherein said class generator comprises:

a semantics extractor to derive a set of binary relationships and to identify the entity type of each database file of said database model;

a means to derive said set of classes and their said set of relationship names using said set of binary relationships and said entity type of each database file of said database model.

41. An end-user query facility as in claim 40 wherein said class generator further comprises a means for a user to modify said relationship names in the said set of classes.

42. An end-user query facility as in claim 6, 7, 18 or 19 wherein said application comprises an interface that displays all or selected said classes and that allows a user to formulate a query by picking the desired class attributes from said class as a designation of the information to be extracted from said database.

43. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 which further comprises a model purifier for a user to alter said database model by defining new keys for database files or by altering existing keys of database files of said database model.

44. An end-user query facility as in claim 43 wherein said keys comprise unique or repeating keys.

45. An end-user query facility as in claim 43 wherein said keys comprise primary or foreign keys.

46. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 which further comprises a model purifier for a user to alter said database model by defining new item(s), new key(s) or new file(s) using items of database files of said database model.

47. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 which further comprises a program analyzer for analyzing and deriving new item(s), new key(s) or new file(s) from source code of applications that access said database model, and for altering said database model using said new item(s), new key(s) or new file(s).

48. An end-user query facility as in claims 46 or 47 wherein said program generator comprises means to generate source program that produce data from said new items of said new files and wherein said end-user query facility further comprises a compiler for compiling said source program.

49. An end-user query facility as in claim 43, 46 or 47 wherein said class generator further comprises a means to read said altered database model to regenerate said set of classes and said set of linkages.

50. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 which further comprises a security model specifier to allow a user to input a security model which specifies restrictions on the information a user can obtain from said database.

51. An end-user query facility as in claim 50 wherein said security model comprises:

item securing which specifies a set of database files and items selected from said database which a user or class of users can access; and

value security which specifies a set of conditions to restrict said user or class of users to a certain range(s) of values of said database.

52. An end user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said program comprises a source code program.

53. An end user query facility as in claim 52 which further comprises a compiler for compiling said source code program.

54. An end user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said application comprises:

means for obtaining from a user a user supplied keyword indicative of the data desired to be extracted from said database;

means for determining all keywords in said database model having a predefined relationship with said user supplied keyword; and

means for causing a user to select one or more of said keywords having a predefined relationship, said one or more keywords thus selected serving as said designation of the information to be extracted from said database.

55. An end-user query facility as in claims 6, 7, 10, 18 or 19 wherein said inference engine comprises:

means for inferring new acquired knowledge threads and storing said new acquired knowledge threads in said knowledge base; and

means for determining an access path to said database using either said basic knowledge threads or said acquired knowledge threads as required to meet the user query.

56. An end-user query facility as in claim 55 wherein said acquired knowledge thread is derived through a combination in parallel of two or more of said basic knowledge threads such that one of the said basic knowledge threads has one or more of its consecutive files in common with the corresponding number of consecutive files starting from the thread head of another of said basic knowledge threads.

57. An end-user query facility as in claims 56 wherein said thread head is a first file on a said basic knowledge thread.

58. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said knowledge base comprises pre-created knowledge base.

59. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said knowledge base comprises a run-time created knowledge base.

60. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said knowledge base comprises a persistent knowledge base.

61. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said knowledge base comprises a transient knowledge base.

62. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said knowledge base is implemented in data dictionary of said database.

63. An end-user query facility as in claims 1, 6, 7, 10, 18 or 19 wherein said knowledge base is implemented in system catalog of said database.

64. An end-user query facility as in claims 10, 18 or 19 which further comprises a second connectivity driver for connecting said program generator to said database through a database connectivity driver.

65. An end-user query facility as in claims 10, 18 or 19 wherein said second e-mail agent further comprises means to post said desired information extracted from said database to the user mailbox.

66. An end-user query facility as in claims 18 or 19 which further comprises a second connectivity driver for connecting said program generator to said database through a database connectivity driver.

67. An end user query facility as in claim 66 wherein both said second connectivity driver and said database connectivity driver have the same data access interface that conforms to a data access interface standard.

68. An end user query facility as in claims 70 or 67 wherein said data access interface standard is Microsoft Corporation's Open Database Connectivity (ODBC) standard.

69. An end user query facility as in claims 70 or 67 wherein said data access interface standard is Apple Computer Incorporated's Data Access Language (DAL) standard.

70. An end-user query facility as in claims 18 or 19 wherein both said application and said first connectivity driver have the same data access interface that conforms to a data access interface standard.

71. An end user query facility for accessing an existing database having a plurality of database files formed using a database model, comprising:

an entity relationship (ER) model generator for reading said databases and deriving a plurality of entity-relationship models, said ER model generator comprising:

an entity type classifier for classifying each said database file into one of a plurality of entity types, said entity types including a "kernel" entity type, a "subtype" entity type, a "characteristic" entity type, an "associative" entity type, and a "pure lookup" entity type;

a binary relationship generator for generating a plurality of binary relationships between said database files, each said binary relationship associated with a linkage representing a relation between a first file having a key that references an equivalent key of a second file, said binary relationships including:

a "has_children" type that represents a first database file that is not classified as a "pure lookup" entity type and has a one-to-many relationship with a second database file, and said second database file does not have a unique key;

a "has_wards" type that represents a first database file that is not of a "pure lookup" entity type and has a one-to-many relationship with a second database file having a unique key;

a "has_subtype" type that represents a first database file having a one-to-one relationship with a second database file and said second database file is classified as a "subtype" entity type; and

an "inverse_of_pure_lookup" that represents a first database file is classified as a "pure_lookup" entity type;

a model constructor for constructing said entity-relationship models, each said entity-relationship model representing a tree having a root and a plurality of branches, said root associated with one of said database files classified as a "kernel" entity type, said model constructor utilizing said entity types and said binary relationships to associate one or more branches with each said entity-relationship model;

a knowledge base that stores said ER models;

an application for interfacing with a user to obtain from said user choices based on said entity-relationship model as a designation of the information to be extracted from said database;

an inference engine that, based upon said designation of information to be extracted from said database, identifies one or more of said database files that contain the desired information and searches said knowledge base to determine one or more linkages connecting said identified files;

a program generator that accesses said linkages obtained by said inference engine and generates a program to extract said desired information from said database; and

a query facility connectivity driver that comprises:

a first connectivity interface coupled to said application that conforms to Microsoft open database connectivity (ODBC) standard to interface with said inference engine and said knowledge base; and

a second connectivity interface coupled to said inference engine and said knowledge base to interface with said existing database using Microsoft open database connectivity (ODBC) driver.

72. An end user query facility as in claim 71 wherein said binary relationships further comprises:

an "inverse_of_has_children" type that represents a second database file that is not classified as a "pure_lookup" entity type and has a one-to-many relationship with a first database file having a unique key;

an "inverse_of_has_wards" type that represents a first database file that is not classified as a "pure lookup" entity type and has a one-to-many relationship with a second database file having a unique key;

a "pure_lookup" type wherein said second database file is classified as a "pure lookup" entity type; and

an "inverse_of_has_subtype" type wherein said second database file has a one-to-one relationship with said first database file, and said first database file is classified as a "subtype" entity type.

73. An end-user query facility as in claim 71 wherein said information scout comprises an interface that displays all or selected ones of said entity-relationship models and that allows a user to formulate a query by picking desired attributes from said entity-relationship model as a designation of the information to be extracted from said database.

74. An end-user query facility as in claim 71 which further comprises a model purifier for a user to alter said database model by defining new keys for database files or by altering existing keys of said database files.

75. An end-user query facility as in claim 71 which further comprises a model purifier for a user to alter said database model by defining one or more new items, one or more new keys or one or more new files using items of said database files.

76. An end-user query facility as in claim 71 which further comprises a program analyzer for analyzing and deriving one or more new items, one or more new keys or one or more new files from source code of applications that access said database model, and for altering said database model using one or more of said new items, one or more of said new keys, or one or more of said new files.

77. An end-user query facility as in claims 75 or 76 wherein said program generator comprises means to generate a source program that produce data from said new items of said new files and wherein said end-user query facility further comprises a compiler for compiling said source program.

78. An end-user query facility as in claim 71 which further comprises a security model specifier to allow a user to input a security model which specifies restrictions on the information a user can obtain from said existing database.

79. An end-user query facility as in claim 78 wherein said security model comprises:

item security which specifies a set of database files and items selected from said database which a user or class of users can access; and

value security which specifies a set of conditions to restrict said user or class of users to one or more ranges of values of said database.

80. An end user query facility as in claim 71 wherein said program comprises a source code program.

81. An end-user query facility as in claim 80 which further comprises a compiler to compile said source code program.

82. An end-user query facility as in claim 71 which further comprises a means for a user to modify relationship names in the said set of entity-relationship models.

* * * * *

US005664173A

United States Patent [19]

Fast

[11] Patent Number: 5,664,173

[45] Date of Patent: Sep. 2, 1997

[54] **METHOD AND APPARATUS FOR GENERATING DATABASE QUERIES FROM A META-QUERY PATTERN**

[75] Inventor: **Ronald Wayne Fast**, Bellevue, Wash.

[73] Assignee: **Microsoft Corporation**, Redmond, Wash.

[21] Appl. No.: **562,916**

[22] Filed: **Nov. 27, 1995**

[51] **Int. Cl.**[6] **G06F 17/30**

[52] **U.S. Cl.** **395/604**; 395/751

[58] **Field of Search** 395/600, 161, 395/603, 604, 605, 601, 751; 364/300, 419

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,763,474	10/1973	Freeman et al.	340/172.5
4,506,326	3/1985	Shaw et al.	364/300
4,688,195	8/1987	Thompson et al.	364/300
5,031,124	7/1991	Bosinoff et al.	364/551.01
5,133,068	7/1992	Crus et al.	395/600
5,197,005	3/1993	Shwartz et al.	364/419
5,315,709	5/1994	Alston, Jr. et al.	395/600
5,347,647	9/1994	Allt et al.	395/575
5,386,550	1/1995	Hedin et al.	395/600
5,519,859	5/1996	Grace	395/600
5,528,748	6/1996	Wallace	395/183.01
5,537,590	7/1996	Amado	395/600
5,550,971	8/1996	Brunner et al.	395/161
5,574,898	11/1996	Leblang et al.	395/601

OTHER PUBLICATIONS

Teach Yourself Web Publishing with HTML in a Week, Laura Lemay, pp. 1–6.

Microsoft Press Computer Dictionary 2nd Edition, Woodcock et al., pp. 115–116.

White et al, Test MAnager: A Regression Testing Tool, IEEE, pp. 338–347 Sep. 1993.

Paulley et al, Exploiting Uniqueness in Query Optimization, IEEE, pp. 68–79 Feb. 1994.

Stepheson et al, Imacts: An Interactive, Multiterabyte Image Archive, pp. 146–161 Sep. 1995.

Primary Examiner—Thomas G. Black

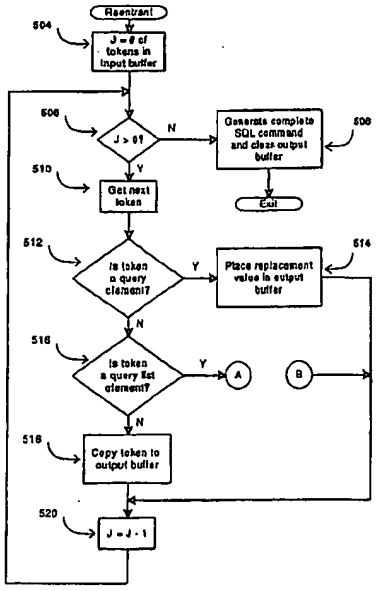
Assistant Examiner—Frantz Coby

Attorney, Agent, or Firm—Duft, Graziano & Forest, P.C.

[57] **ABSTRACT**

A grammar, parsing method, and associated apparatus for automatically generating test commands to test an SQL database engine interface while reducing storage requirements and improving access time for such test commands as compared with prior test tools. The test tools and methods include a grammar for concise syntactic representation of a meta-query (also referred to as meta-language statement, query pattern, or query template). The meta-query defines an statement similar to the SQL language but includes query elements and query list elements used to generate a plurality of SQL test commands to be applied to the SQL database engine under test. Test commands are generated from the meta-query to reduce storage requirements of prior test methods. Query elements are variable space holders in the meta-query and are replaced by a value appropriate to the SQL database engine under test when the meta-query is used to generate test commands. Query list elements define a list of values to be inserted in place of the query list element when generating the test commands from the meta-query.

9 Claims, 7 Drawing Sheets



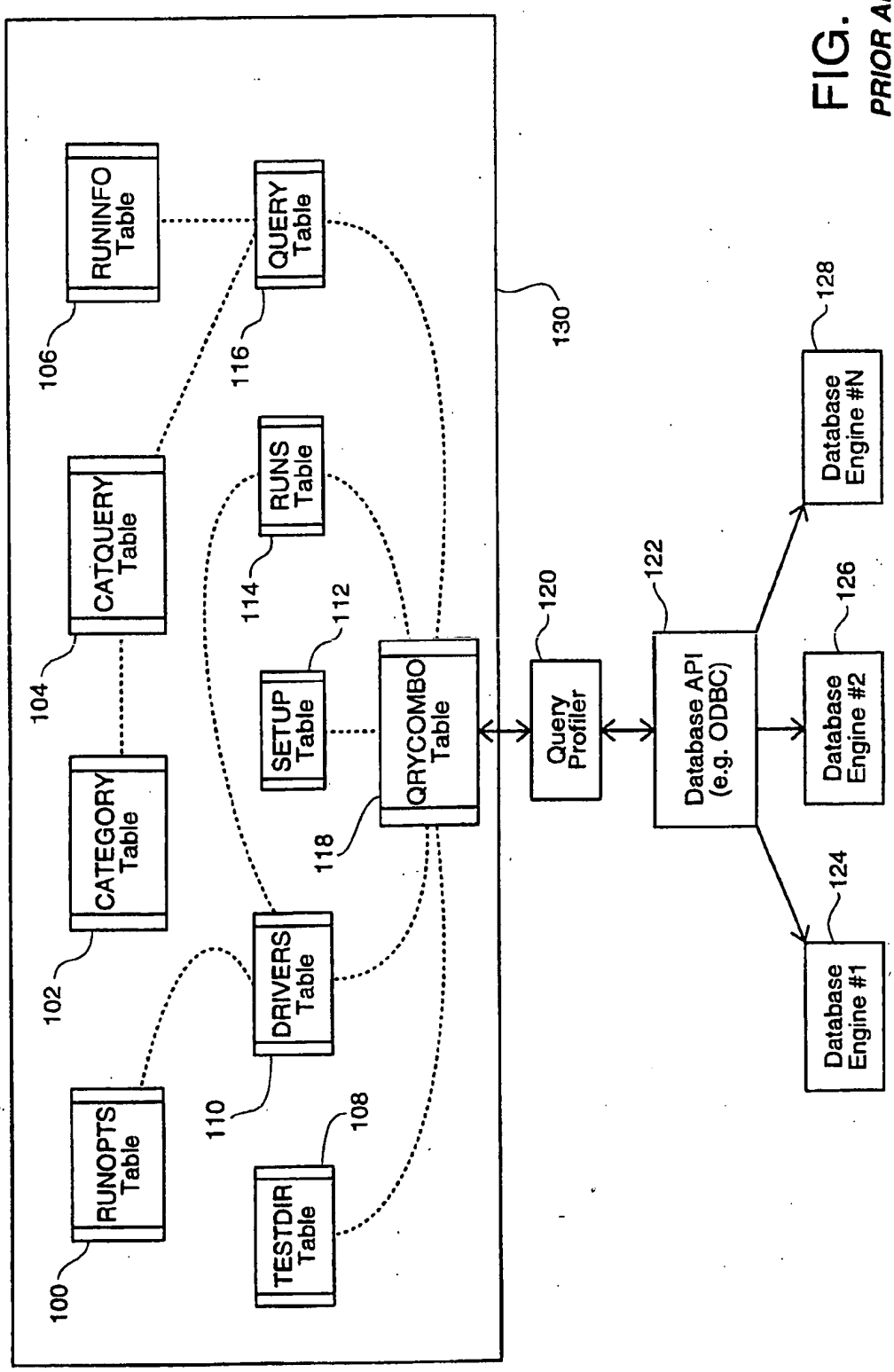


FIG. 1
PRIOR ART

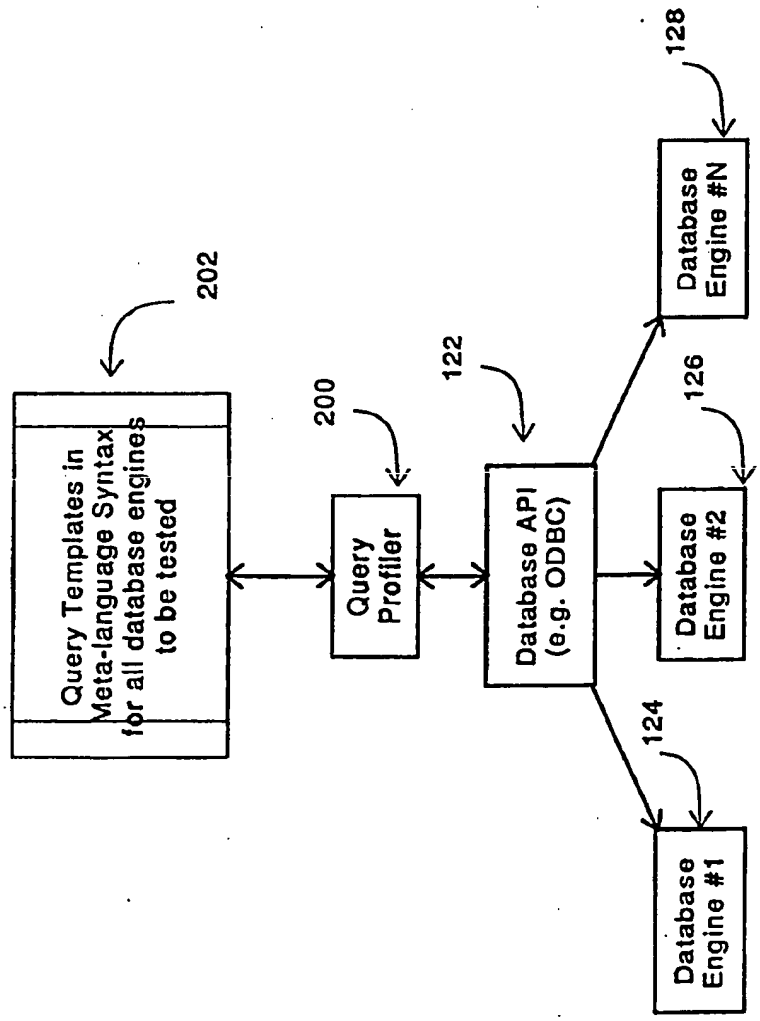

Query Templates in Meta-language Syntax for all database engines to be tested
202
Query Profiler
200
Database API (e.g. ODBC)
122
Database Engine #N
128
Database Engine #2
126
Database Engine #1
124

FIG. 2

FIG. 3

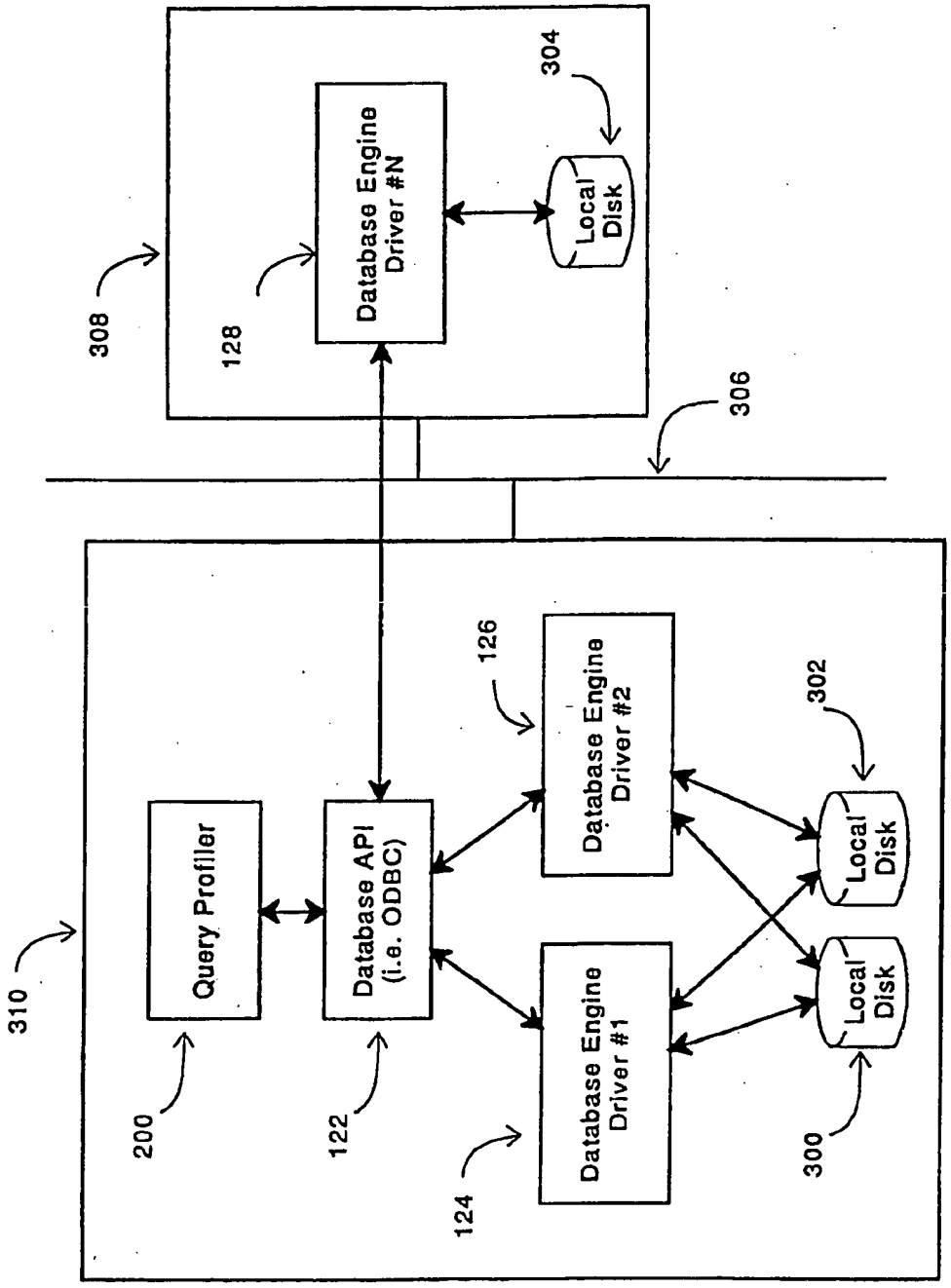
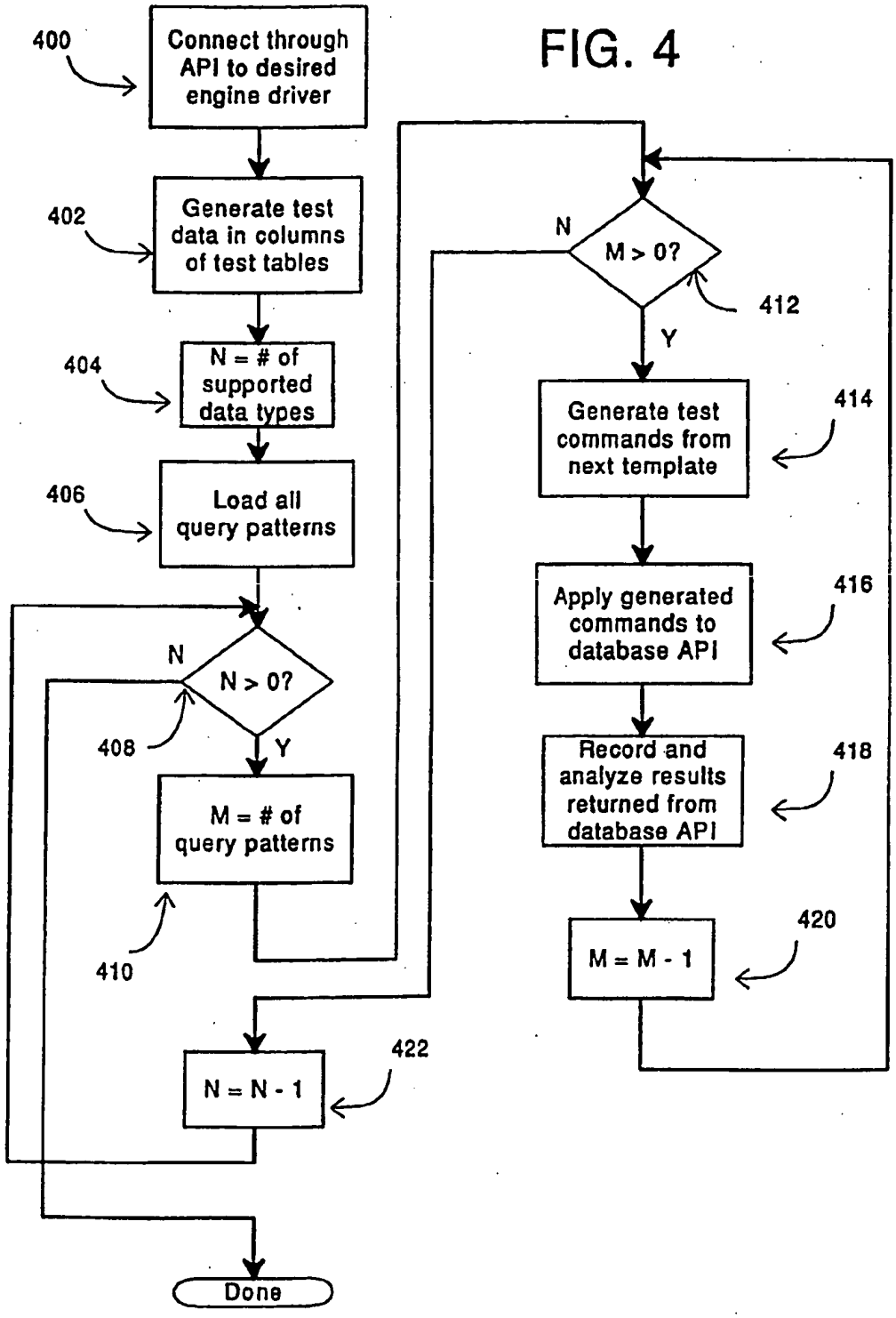

308
128
Database Engine Driver #N
304
Local Disk
306
310
Query Profiler
200
Database API (i.e. ODBC)
122
Database Engine Driver #2
126
Database Engine Driver #1
124
302
Local Disk
Local Disk
300

FIG. 4


400
Connect through API to desired engine driver
402
Generate test data in columns of test tables
404
N = # of supported data types
406
Load all query patterns
N
N > 0?
408
Y
M = # of query patterns
410
N
M > 0?
412
Y
Generate test commands from next template
414
Apply generated commands to database API
416
Record and analyze results returned from database API
418
M = M - 1
420
N = N - 1
422
Done

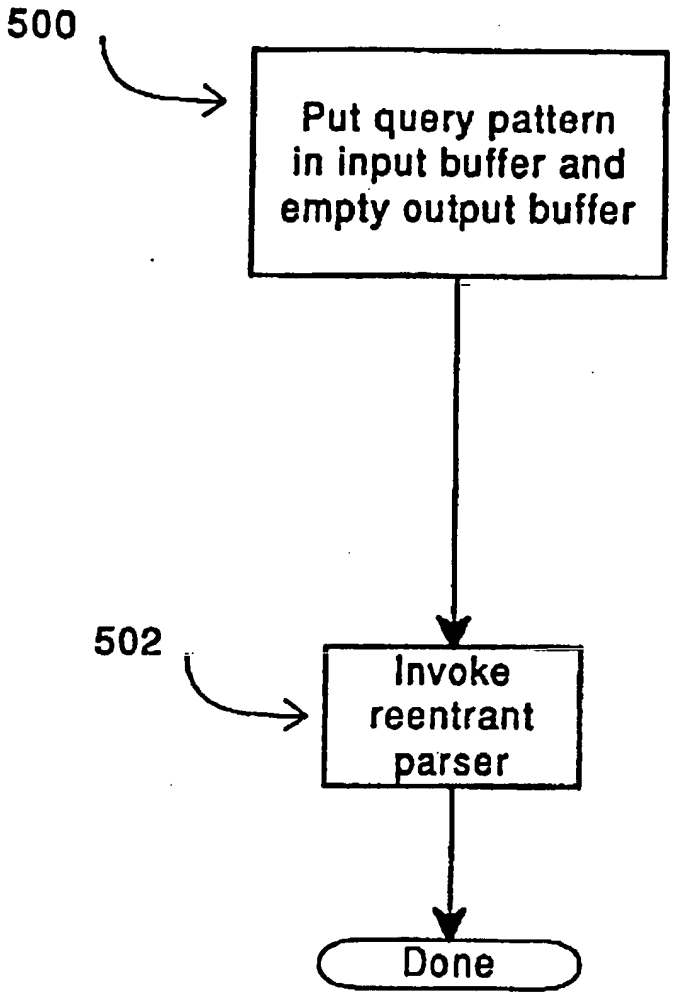

500
Put query pattern in input buffer and empty output buffer
502
Invoke reentrant parser
Done

FIG. 5

FIG. 6

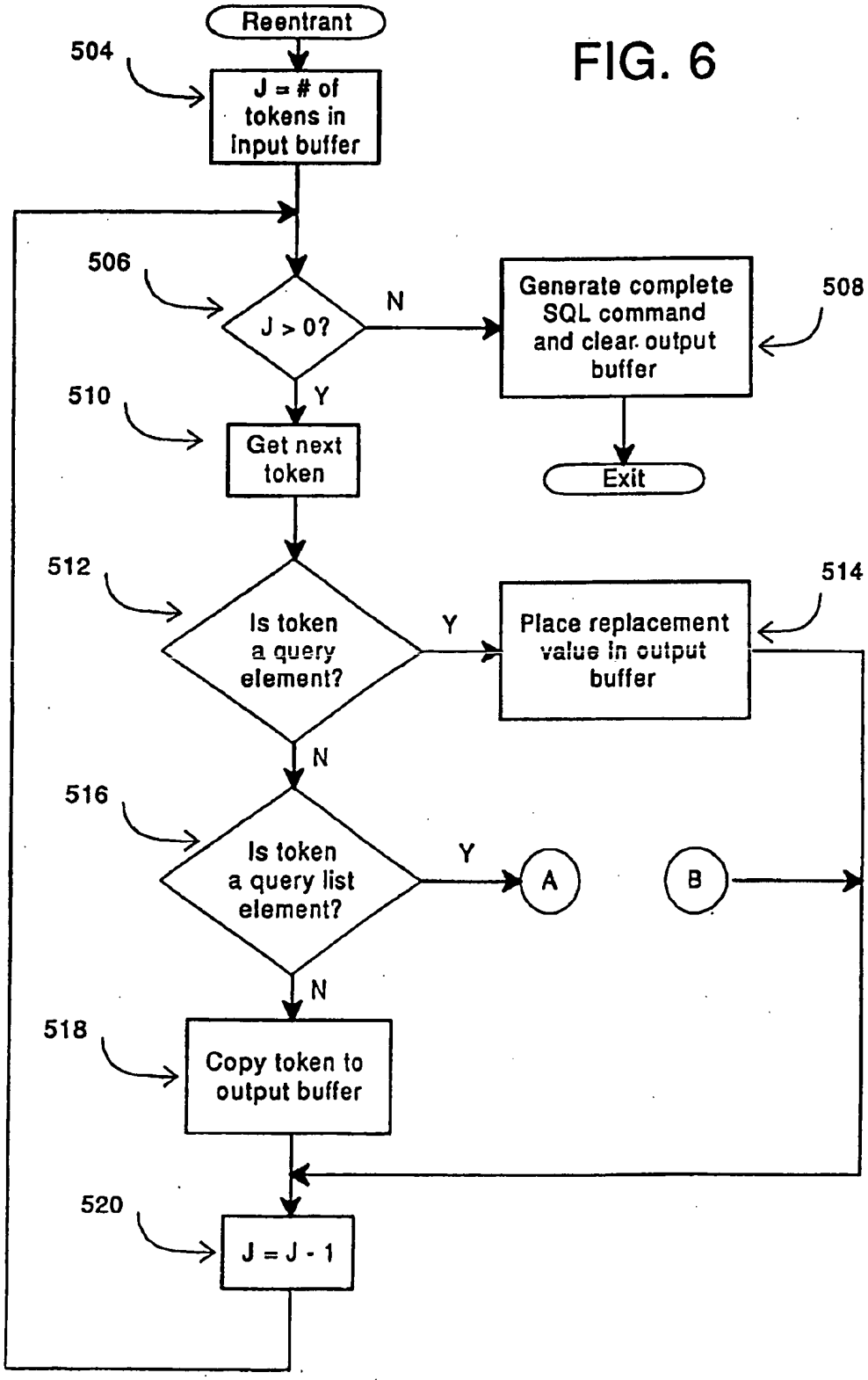
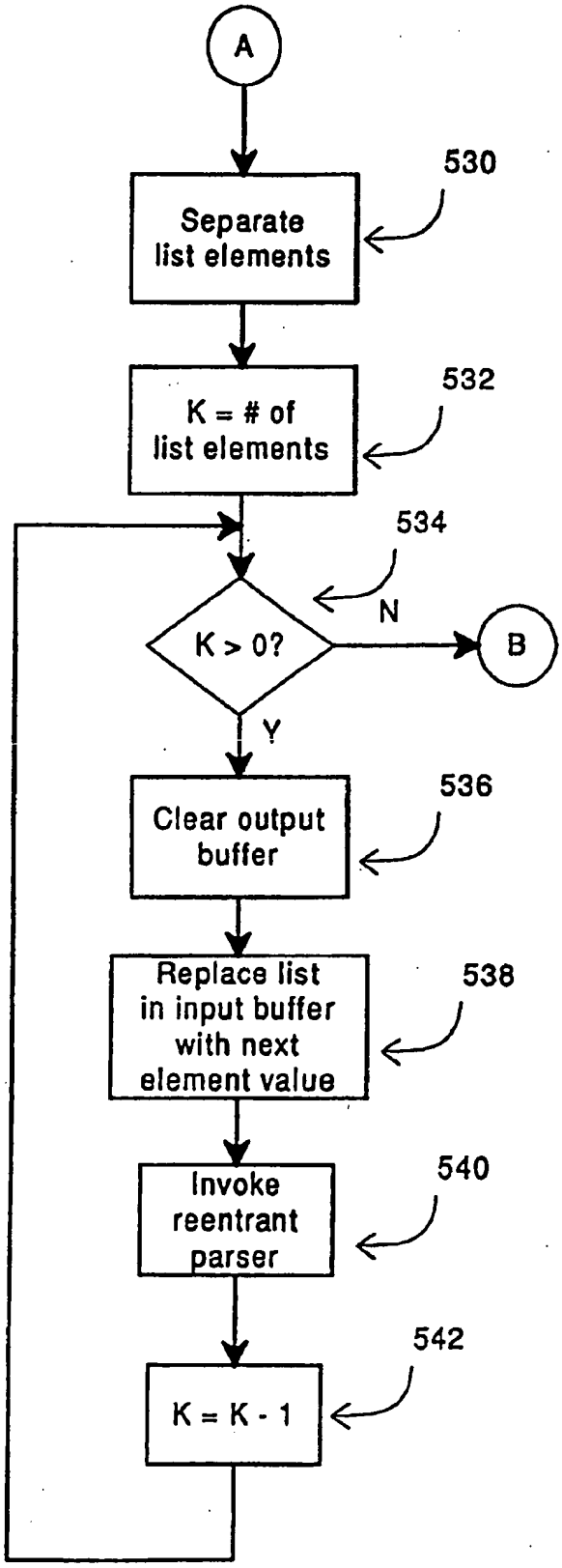

Reentrant
504
J = # of tokens in input buffer
506
J > 0?
N
Generate complete SQL command and clear output buffer
508
Y
510
Get next token
Exit
512
Is token a query element?
Y
Place replacement value in output buffer
514
N
516
Is token a query list element?
Y
A
B
N
518
Copy token to output buffer
520
J = J - 1

FIG. 7


A
530
Separate list elements
532
K = # of list elements
534
K > 0?
N
B
Y
536
Clear output buffer
538
Replace list in input buffer with next element value
540
Invoke reentrant parser
542
K = K - 1

METHOD AND APPARATUS FOR GENERATING DATABASE QUERIES FROM A META-QUERY PATTERN

FIELD OF THE INVENTION

This invention relates to the testing of database software systems and in particular to the testing of database engine drivers in an Open DataBase Connection (ODBC) database environment by the automatic generation of test commands from a meta-query pattern.

PROBLEM

The computing structures and methods of the present invention are built upon open standard database Application Program Interfaces (APIs—also referred to herein as database management interface means) such as Microsoft's ODBC or X/Open's DATA MANAGEMENT: SQL CALL LEVEL INTERFACE (X/Open Preliminary Specification P303 ISBN 1-85912-015-6 - was previously publication S203 - will become publication C451 available from X/Open Company Ltd, Berks, United Kingdom). These standards permit client/server database application programs to be designed in accord with a common, standardized API while utilizing any underlying database engine which conforms to these standards for the permanent physical storage of the managed information. End user installations using the present invention may therefore utilize any presently installed database management subsystem. The SQL (Structured Query Language) has been widely adopted as a de facto standard interface for the specification of database queries (and related data management commands). The ODBC API therefore enforces a standardized SQL query language and performs any translations necessary for operation of a query upon a specific database engine (database management subsystem). This hierarchical API structure permits the application programmer to adhere to a single database/query architecture and yet easily adapt (port) the application program to the unique requirements of a particular database engine through the ODBC API library functions.

In testing such a standard database API, a test process must generate a large number of test commands for each database engine supported by the API. For example, a large set of test commands is applied to Microsoft's ODBC API in order to test its use in conjunction with the dBase database engine. Yet another large set of test commands is needed to test ODBC when used in conjunction with the Access or Paradox database engines, etc. Though there is substantial similarity in these plural sets of test commands, there are invariably minor differences in syntax or semantics between the queries generated for each unique database engine. For example, some database engines support atomic data types which are unique to the engine. Or, for example, the size limits for certain data types may vary among various database engines. In view of these differences, prior test methods and tools for generating test commands for database API subsystems have created large sets of test commands and commands and stored them in a query database to be retrieved when the corresponding database engine is tested with the ODBC API. Each test command is "hard-coded" for the specific database engine to which it corresponds. The query database which stores these commands can therefore be quite large. As such, as with any large database, access to the database for purposes of extracting test commands to perform a particular test sequence can be quite time consuming. Adding, deleting or modifying test commands stored in the large query database can also be time consuming due to re-indexing operations associated with the changes in the query database.

An additional problem with the query database techniques taught by prior test products and methods arises from the fact that the query database is itself another database which must be operated in the same computing platform on which the test commands are being applied to the ODBC API. Whatever DBMS package is used for the storage of the test commands in the query database must be available on, or ported to, the computing platform on which the ODBC/database engine combination is being tested. This porting effort may add a substantial workload to the ODBC test efforts if the DBMS selected for the query database storage is not presently available on the computing platform presently being tested.

In view of the above discussions, it is dear that there exists a need for methods and apparatus for managing and manipulating test commands to be used in testing an ODBC/database engine combination which improves speed of access to the test commands, eases the modification of the commands, and reduces the storage requirements for the storage of the test commands.

SOLUTION

The present invention solves the above identified problems and other problems to thereby advance the state of the useful arts by providing methods and associated apparatus for generating SQL test commands from a query pattern (also referred to herein as query template, meta-query, or simply meta-language statement). The query pattern is formed according to the syntax of a meta-language of the present invention to define a set of SQL test commands in a concise syntactic statement. Each meta-language test command pattern (a meta-query) is parsed by the methods of the present invention to generate all test commands in the set defined by the meta-query. The SQL test commands so generated are then applied to the database engine under test.

The meta-language of the present invention permits test commands to be expressed in a concise, compact meta-language syntax. Storage and modification of the concise, compact meta-queries is simpler, faster, and requires significantly less storage capacity as compared to the prior techniques wherein all individual test commands are stored in a test database. The meta-queries are stored in a standard text file and may therefore be accessed or modified by any of several well known techniques for viewing and modifying text files.

The meta-language of the present invention expresses the meta-queries according to the rules of a grammar definition. The grammar definition includes "query elements" and "query list elements." The query elements serve as variable place holders in the SQL test commands specified by the meta-query. When the meta-query is processed to generate test commands, the query element placeholder is replaced by a variable value appropriate for the database engine being tested. Query list elements provide a list of values to be substituted into the generated test commands as each test command is generated. When a query list element is specified in a meta-query, at least one query is generated for each element in the query list element. If multiple query list elements are specified in a meta-query, then a test command is generated for each unique combination generated by selecting one of the elements in each of the multiple query, list elements.

The syntax of the meta-language is clearly and completely defined by a simple BNF style specification as compared to

a complex database structure used by prior methods to store and retrieve the set of test commands appropriate to the database engine under test. The BNF definition defines the rules for construction and generation of meta-language commands the semantic interpretation of which is used to generate a set of SQL test commands.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of a typical test environment known in the art for testing a database API (e.g. Microsoft's ODBC);

FIG. **2** is a block diagram of the database API test environment of the present invention which utilizes a meta-language syntax to represent large numbers of test commands;

FIG. **3** is a block diagram of a computing environment in which the test environment of the present invention operates;

FIG. **4** is a flowchart describing the operation of the query profiler in accord with the methods of the present invention;

FIG. **5** is a flowchart which depicts addition detail of the method shown in FIG. **4**;

FIG. **6** is the first half of a flowchart of the reentrant parser of the present invention which generates test commands from meta-language statements; and

FIG. **7** is the second half of a flowchart of the reentrant parser of the present invention which generates test commands from meta-language statements.

DETAILED DESCRIPTION OF THE INVENTION

OVERVIEW:

FIG. **1** is a block diagram of an approach to testing the database API (such as Microsoft's ODBC) in conjunction with a chosen database engine. Query profiler **120** generates test SQL commands and applies the generated test commands to the database API **122** to be tested. The commands generated are intended to test the database API **122** for proper operation in conjunction with one of the plurality of database engines 1 through N (**124**, **126**, and **128**). In accord with the known methods for implementing query profiler **120**, test database **130** is constructed and maintained to contain all possible query commands and associated options for the generation of all test SQL queries applicable to all database engines **124**, **126**, and **128** associated with database API **122**.

The precise structure of test database **130** may be specific to each database API **122** or specific to the needs of the database engines **124**, **126**, and **128** to be used in conjunction with the API **122**. Therefore, the detailed structure of test database **130** is not relevant to an overall understanding of the operation of known prior techniques. Tables **100–118** are intended only as an exemplary database structure to demonstrate the complexity of prior approaches. The various tables and relationships depicted in FIG. **1** are used to define and store the various commands needed to setup a particular ODBC environment for testing a particular ODBC driver, to store the various command options and command parameters, and to store the test commands themselves, among other information. The complex of the test database grows dramatically as additional options, parameters, configurations and environments are added to the testing of each ODBC driver.

FIG. **2** is a block diagram of a query profiler **200** which utilizes the structures and methods of the present invention. Database API **122** and the database engines **124**, **126**, and **128** are identical to those of FIG. **1**. Query profiler **200** of FIG. **2** retrieves and processes the meta-language statements (query templates) from the query templates file **202**. Each meta-language statement in the query templates file **202** may define a plurality of test commands to be generated by the query profiler **200**. The query templates file **202** is a simple text file which may be easily constructed and maintained by any of several well known tools for manipulating text files. The storage space required to store the query templates file **202** is significantly reduced as compared to the storage requirements for equivalent the test database **130** of FIG. **1**.

META-LANGUAGE SYNTAX AND SEMANTICS:

The meta-language of the present invention may be viewed as a set of grammatical rules for constructing statements used by the query profiler **200** of FIG. **2** to generate test SQL commands. The meta-language is substantially similar to the well known SQL query language with elements added to define rules for the construction of actual SQL statements. A typical SQL query command, for example, consists essentially of the following syntax:

```
SELECT column FROM tables WHERE condition
```

where: column is replaced by one or more column names, tables is replaced by one or more table names, and condition is replaced by a logical expression which must evaluate to true for each row to be selected from the tables. The result of the SQL query is a table constructed of the identified columns and the rows selected by virtue of the logical expression evaluating true for those rows. Some database engines provide for additional elements to be named in the identifying columns or tables or in the logical condition expression. For example, parameters which further control the search capability in the engine's data management or specific limitations or additions relating to types of supported data are frequently added to the features of a specific database engine. Such additional elements are frequently unique to the specific database engines supported by the database API. To thoroughly test a database API (such as Microsoft's ODBC) requires testing not only the features common to all supported database engines, but also requires the testing of features unique to each supported database engine. Testing these database engine specific features in conjunction with the database API requires the creation of a large number of specialized command options.

The present invention defines a meta-language syntax and grammar which builds upon the syntax of standard SQL commands. The meta-language syntax adds variable elements to the SQL command syntax. When "parsed" by the query profiler **200** (of FIG. **2**) of the present invention, these variable elements in the meta-language commands are replaced by actual values and the resultant SQL commands are thereby generated from the meta-language statements (without the variable element syntax embedded). The generated SQL commands are then applied to the database API **122** to test its proper operation in conjunction with one of the database engine drivers (**124**, **126**, or **128**). The variable elements of the meta-language statement can specify one or more actual values to use in the generation of test SQL commands and may therefore compactly represent a large volume of generated test SQL commands. Such large volumes of test SQL commands previously required significant mass storage capacity and associated complexity to store and retrieve the several SQL command sets required to test the database API **122** operation.

Processing of the meta-language statements by the query profiler **200** (of FIG. **2**) automatically generates the test SQL

commands represented by the meta-language statements for every data type supported by the specific database engine driver under test (**124**, **126**, or **128** of FIG. 2). The query profiler **200** detects the data types supported by the database engine through standard function calls of the database API **122**. Such interface function calls to the API are well known to those of ordinary skill in the art and are clearly described in the public documentation available with the API (such as Microsoft's ODBC database API). The query profiler **200** then loops through the processing of the meta-language statements to generate test SQL commands once for each supported data type.

The variable element of the meta-language adds "query elements" to the SQL query command syntax which are replaced in generation of the test commands by actual values appropriate to the database engine (**124**, **126**, and **128** of FIG. 1) under test with the database API **122** of FIG. 2. The query elements are identified by query element identifiers (name strings for example) and are delimited in the meta-language statement by angled braces ("<" preceding the query element identifier and ">" following the query element identifier). The following Table 1 provides exemplary query elements presently contemplated in the best known mode of the present invention. One of ordinary skill in the art will readily recognize that this list may be extended to include other query elements which hold the place of language elements in the generated SQL queries and are unique to the database engine drivers.

TABLE 1

Query Element	Replacement Information
<qualifier>	The current qualifier for the database engine driver under test (i.e. the ODBC connection option - SQL_CURRENT_QUALIFIER)
<tableN>	Name of a table in test data for the SQL command (where N is a number from 1 through the number of tables in the test data)
<columnN>	Name of a column in a table in the test data for the SQL command (where N is a number from 1 through the number of columns in the associated table)
<alias>	Name of an alias for a column in the test data for the SQL command
<data>	A constant data value for use in the SQL command
<column name>	Name of generated column (i.e. one that doesn't currently exist in the created table and used in the "ALTER TABLE" queries so that the column name won't conflict with an existing column name)
<column def >	Data type of the <column name> element

The meta-language (query templates/query patterns) of the present invention also includes "query list elements" which, when used in a meta-language statement, cause the generation of a plurality of SQL commands; one for each element in the query element list. This feature of the meta-language permits the compact representation of a large set of test commands in a concise, single, meta-language statement. This representation of a collection of test commands is simpler to maintain or modify and requires significantly less storage than the methods employed in the past to test a database API.

A query list element provides a list of alternate values to be used in generating test commands from the meta-language statement (query template). Each of the alternate values in the query list element is used to replace the query list element in the generation of one (or more) test commands. In other words, a query list element that indicates four alternate values will generate (at least) four test commands, (at least) one each for each of the four alternate values in the query list element. If multiple query list elements are included in a meta-language statement, then the query profiler (**200** of FIG. 2) will generate a test command for each combination of the elements in all the query list elements of the statement.

Query list elements are replaced in generation of the test command by the alternate values supplied in the query list element when test commands are generated to test the database API **122**. The query list elements are comma separated values delimited by a pair of square braces (a "[" preceding the list and a "]" following the list). The following Table 2 provides exemplary query list elements presently contemplated in the best known mode of the present invention. One of ordinary skill in the art will readily recognize that this exemplary list may be extended to include other query list elements which hold the place of language elements in the generated SQL queries.

TABLE 2

Query List Element	Replacement Information
[=,<,<=,>,>=,!=,!<,!>]	Generates eight test commands; one with each of the eight listed logical test (comparison) operators (as used in the condition clause)
[*=,=*]	Generates two test commands; one with each of two Microsoft SQL Server syntax outer join operators
[-,+,*,/,%]	Generates five test commands: one with each of the five listed arithmetic operators
[SQL_DATE, SQL_TIME_STAMP]	Generates two test commands: one with each of the two listed standard data types

The following meta-language statement examples provide further clarification of the power and syntax of the meta-language for the specification of large numbers of test SQL commands. In particular it is to be noted that the meta-language may be applied to many SQL commands (not merely the "SELECT" command).

```
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table>.<column1> [=,<,<=,>,>=,!=,!<,!>] <table2>.<column1>
```

This exemplary meta-language statement generates eight queries selecting rows from column 1 of table 1 (in the test data) where the column 1 value in table 1 of each row compares using the selected one of eight comparison operators with the same row and column of table 2. The eight generated queries are:

```
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table1>.<column1> = <table2>.<column1>
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table1>.<column1> < <table2>.<column1>
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table1>.<column1> <= <table2>.<column1>
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table1>.<column1> > <table2>.<column1>
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table1>.<column1> >= <table2>.<column1>
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table1>.<column1> != <table2>.<column1>
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table1>.<column1> !< <table2>.<column1>
SELECT <table1>.<column1> FROM <table1>,<table2> WHERE
<table1>.<column1> !> <table2>.<column1>
```

In addition, the query profiler **200** of FIG. 2 will generate these eight test commands for all data types supported by the selected database engine driver (**124**, **126**, or **128** of FIG. 2). For example, Microsoft Access version 2.0 supports 15 distinct data types. Therefore, this exemplary meta-language

statement generates 8*15 or 120 test commands when testing the database API **122** in conjunction with a Microsoft Access database engine driver.

As a further example, consider:

```
CREATE INDEX <table 1 > ON <table 1 > ( <column name>
ASC ) WITH IGNORE NULL
```

This exemplary meta-language statement generates a new table index with a column name appropriate to the data type currently being processed by the query profiler **200**. As noted above, Microsoft Access, for example, supports 15 data types and therefore, this meta-language statement generates 15 SQL commands when testing the database API **122** in conjunction with the Microsoft Access database engine driver.

QUERY PROFILER:

Query profiler **200** of FIG. **2** is operable on a data processing system to parse the meta-language statements and to generate test SQL commands for application to the database API **122**. FIG. **3** is a block diagram depicting a typical computing environment in which query profiler **200** operates. Data processing system **310** provides the central processing, memory, and mass storage components for operation of query profiler **200**, database API **122**, and database engine drivers **124** and **126**. Database engine drivers **124** and **126** store and retrieve information on local disks **300** and **302**. Data processing system **310** may be connected to other data processing systems **308** over network attachment **306**. Additional database engine drivers **128** and local disks **304** may reside within the data processing system **308**. Database API **122** may interact with a remote database engine driver **128** through any of several well known network computing architectures. Further, one of ordinary skill in the computing arts will readily recognize that the computing environment depicted in FIG. **3** is only exemplary of one such architecture in which the structures and methods of the present invention may operate. The present invention is equally applicable to computing environments without networked connections to other data processing system or to distributed computing environment utilizing other topological configurations or connectivity technologies.

FIG. **4** is a flowchart depicting the methods of the present invention as implemented by the query profiler **200**. Element **400** of FIG. **4** invokes functions in the database API (**122** of FIG. **2**) required to associate the test procedure with a particular database engine driver module under test (**124**, **126**, or **126** of FIG. **2**). Element **402** then generates test data in tables created and managed by the database engine driver under test. This test data is used by the selected database engine **124**, **126**, or **128** through the database API **122** at the direction of the query profiler **200** in its interpretation of the meta-language statements. Since the query profiler generates the test data, it can predict the expected result of each SQL command generated from the meta-language statements and applied to the database API and engine. The specific form of the generated tables is a matter of design choice made by the test engineering staff in creating the test procedures. One or more tables may be created and each table may have one or more columns as desired by the test engineers to adequately test the database API interface to the database engine driver.

Elements **404** and **406** initialize for the looping functions performed by elements **408–420**. The test SQL commands generated for testing the API interface to the engine are generated for each data type supported by the underlying database engine. Element **404** sets the variable "N" to the number of data types supported by the selected database engine. Element **406** loads all the query patterns from a text file in which they are stored. The query patterns are previously designed by the test engineers to compactly specify the voluminous test commands required to adequately test the interface between the database API and a database engine driver module. As discussed above, the query patterns are written in simple textual form in the syntax of the meta-language discussed above. Element **406** serves to read the text file storing the pre-defined query patterns in preparation for parsing the meta-language statements and generating the specified SQL commands therein.

Elements **408–422** are repetitively operable for each data type supported by the selected database engine driver. Element **408** tests whether the counter variable "N" (indicating the number of supported data types) has been decremented to zero. On each iteration of the loop (elements **408–422**), element **422** is operable to decrement the counter variable "N." Elements **410–420** are therefore operable to generate the test commands specified by all query patterns for a single data type supported by the selected database engine driver.

Element **410** sets the variable "M" to the number of query patterns pre-defined by the test engineers in the text file. In other words, the number of records to be processed in the meta-language file. Each record provides another query pattern in the meta-language syntax described above. Each record is therefore processed in turn to generate all the test SQL commands required to test the database API in conjunction with the selected database engine driver.

Elements **412–420** are repetitively operable for each record (meta-language statement or query pattern) retrieved from the text file. Element **412** tests whether the counter variable "M" (indicating the number of meta-language statements in the text file) has been decremented to zero. On each iteration of the loop (elements **412–420**), element **420** is operable to decrement the counter variable "M." Elements **414–418** are therefore operable to generate the test commands specified a single query patterns for a single data type supported by the selected database engine driver.

Element **414** parses the meta-language statement to process all query elements and query list elements. Parsing of the meta-language statement includes locating all query elements and replacing them by values appropriate to the particular data type presently being processed and as appropriate for the selected database engine driver. Additionally, the parsing process locates any query list elements in the meta-language statement and generates one SQL command for each element in the list. Each of the generated SQL commands are thereby generated by substitution of actual values for the variable elements of the meta-language statement.

Element **416** then applies the SQL commands generated by element **414** to the database API **122**. The SQL commands so applied are in turn transformed and transferred to the selected database engine driver **124**, **126**, or **128** of FIG. **2** for actual processing upon the test data stored on the mass storage devices (**300**, **302**, and **304** of FIG. **3**). Element **418** captures, records, and analyzes the results of the SQL command processing returned by the database engine driver. Processing of these results is discussed below in additional detail.

As noted above, element **420** is next operable to decrement the Loop counter variable "M" and element **422** decrements the loop counter variable "N" to control the iterative looping of the method. When element **412** determines that all records in the meta-language text file have been processed, it returns control to element **422** to process another supported data type. Likewise, when element **408**

determines that all supported data types have been processed, the method completes processing.

FIGS. 5–7 combine to provide a flowchart providing additional detail of the operation of element 414 of FIG. 4 which generates all SQL commands from a single query template (meta-language statement). Element 500 of FIG. 5 places the query pattern (meta-language statement) to be parsed into a memory input buffer. Element 502 of FIG. 5 then initially invokes the reentrant parser to parse the tokens of the meta-language statement. Tokens in the meta-language statement (query pattern or template) are, in their simplest form, fields of non-space characters separated by spaces. Each token is therefore either a query element (if it is delimited by angle braces), or a query list element (if it is delimited by square braces), or is a constant textual string which forms a constant portion of the desired SQL command to be generated. There may be a plurality of query elements or query list elements in a single meta-language statement. In addition, the elements of a query list element may themselves be other query elements or query list elements (i.e. nested variable portions of the query template). For this reason, the parser of the query profiler of the present invention is reentrant so as to permit parsing of nested variable elements within the template.

FIG. 6 depicts the details of the reentrant parser of the query profiler. The parser is entered in a reentrant manner: i.e. saving previous status and allocating local variables on a stack. Element 504, sets the counter variable "J" to the number of tokens found in the input buffer counter varible J is provided as a parameter to the reentrant function. Elements 506 and 520 are operable to loop on the invocation of elements 510–518 (and 530–542 of FIG. 7 below) for each token found in the input buffer. If element 506 determines that all tokens in the input buffer have been processed, element 508 is operable to generate the completed SQL command in the output buffer. The completed command is then applied to the database API (122 of FIG. 2) as discussed above with respect to FIG. 4. If further tokens remain to be processed, element 510 is operable to get the next token from the input buffer for further processing.

Element 512 determines if the token to be processed is a query element type of token (i.e. delimited by angle braces). If so, element 514 is operable to copy the replacement value for the query element (as discussed above) into the output buffer. This replacement value stands in place of the query element in the SQL command being generated from the query template. Processing then continues at element 520 by looping through the process.

If the token is not a query element, the element 512 determines whether the token is a query list element (i.e. delimited by square braces). If not, the token must be a constant portion of the query pattern and so is simply copied to the output buffer to become a constant part of the generated SQL command. If the token is a query list element, processing continues at element 530 of FIG. 7.

Element 530 of FIG. 7 separates the query list elements into the individual values (the comma separated values of the list). Element 532 sets the counter variable "K" to the number of value elements in the list. If element 534 determines that there are no more values in the list to be processed, then processing continues by returning to element 520 of FIG. 6.

For each value in the list, elements 534–542 are invoked to generate an SQL command in the output buffer. Element 536 first clears the output buffer generated up to this point (by earlier operation of elements 506–520 of FIG. 6). Next, element 538 creates a new input buffer with the current input buffer but with the query list element (now being processed) replaced by the next value from the list Element 540 then invokes the reentrant parser function to re-parse the new input buffer with the currently processed query list element replaced by its next value from the list. After processing of the revised meta-language statement (the new input buffer) is complete, and the associated SQL commands are generated, processing continues in the present invocation of the parser with element 542 decrementing the loop count variable "K" to indicate another value in the list is processed. Upon completion of the processing of the present query list element, processing continues at element 520 if FIG. 6 to process the remaining tokens of the meta-language statement.

Processing continues in this manner for each value in the query list element until all SQL commands represented by the query pattern (meta-language statement) have been generated. One of ordinary skill in the art will recognize that other forms of recursive of reentrant designs of the method of the present invention may achieve the same purpose. Such design choices for reentrant or recursive methods are well known to those of ordinary skill in the software arts. In addition, the methods of the present invention may be simplified by restricting the meta-language syntax to prohibit the nesting of, or even a plurality of, query list elements. Such a design choice eliminates the need for recursion in the processing of the meta-language. Again, such design choices are well known to those of ordinary skill in the software arts.

BNF DESCRIPTION OF GRAMMAR RULES:

The meta-language of the present invention may be understood as a set of grammatical rules for the formation of legal statements within the grammar. A BNF format description is a common format in which to express such rules. The following BNF rule description includes the entire SQL standard language grammatical rules from which the rules of the present invention are an extension. The extensions to the SQL grammar defined by the rules of the present invention are highlighted in bold characters to distinguish them from the standard rules which comprise the standard SQL language. For added clarity, the enhancements to the SQL BNF grammar rules all have identifiers that begin with the characters "QP".

Elements used in SQL Statements:

all-function ::= {AVG | MAX | MIN | SUM} (*expression*)
approximate-numeric-literal ::= *mantissa*E*exponent*
approximate-numeric-type ::= {approximate numeric types}
argument-list ::= *expression* | *expression*, *argument-list*
base-table-identifier ::= ***QP-base-table-name***
base-table-name ::= *base-table-identifier*
| *owner-name*.*base-table-identifier*
| *qualifier-name qualifier-separator base-table-identifier*
| *qualifier-name qualifier-separator* [*owner-name*].*base-table-identifier*
between-predicate ::=
expression [NOT] BETWEEN *expression* AND *expression*
binary-literal ::= {implementation defined}
binary-type ::= {binary types}
bit-literal ::= 0 | 1
bit-type ::= {bit types}
boolean-factor ::= [NOT] *boolean-primary*
boolean-primary ::= *predicate* | (*search-condition*)
boolean-term ::= *boolean-factor* [AND *boolean-term*]
character ::= {any character in the implementor's character set}
character-string-literal :: = '{*character*}...'
character-string-type ::= {character types}
column-alias ::= ***QP-alias***
column-identifier ::= ***QP-column-identifier***

16

1001/006

column-name ::= [*table-name.*]*column-identifier*
column-name ::= [{*table-name* | *correlation-name*}.]*column-identifier*
comparison-operator ::= < | > | <= | >= | = | <>
comparison-predicate ::= *expression comparison-operator expression*
comparison-predicate ::= *expression QP-comparison-list expression*
comparison-predicate ::= *expression QP-outer-join-list expression*
comparison-predicate ::=
    expression comparison-operator {*expression* | (*sub-query*)}
correlation-name ::= *QP-alias*
cursor-name ::= *QP-cursor-name*
data-type ::= *character-string-type*
data-type ::=
    character-string-type
    | *exact-numeric-type*
    | *approximate-numeric-type*
data-type ::=
    character-string-type
    | *exact-numeric-type*
    | *approximate-numeric-type*
    | *bit-type*
    | *binary-type*
    | *date-type*
    | *time-type*
    | *timestamp-type*
date-separator ::= -
date-type ::= {date types}
date-value ::=
    years-value date-separator months-value date-separator days-value
date-value ::=
    QP-sql-date-time-list
days-value ::= *digit digit*
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
distinct-function ::=
    {AVG | COUNT | MAX | MIN | SUM} (DISTINCT *column-name*)
dynamic-parameter ::= ?
empty-string ::=
escape-character ::= *character*
exact-numeric-literal ::=
    [+|-] { *unsigned-integer*[.*unsigned-integer*]
    | *unsigned-integer.*
    | .*unsigned-integer* }
exact-numeric-type ::= {exact numeric types}
exists-predicate ::= EXISTS (*sub-query*)
exponent ::= [+|-] *unsigned-integer*
expression ::= *term* | *expression* {+|-} *term*
expression ::= *term* | *expression QP-math-operation-list term*

1001/006

factor ::= [+|-]*primary*
hours-value ::= *digit digit*
index-identifier ::= *QP-index-name*
index-name ::= [*index-qualifier.*]*index-identifier*
index-qualifier ::= *QP-index-qualifier*
in-predicate ::= *expression* [NOT] IN {(*value* {, *value*}...) | (*sub-query*)}
insert-value ::=
    dynamic-parameter
    | *literal*
    | NULL
    | USER
keyword ::=
    (see list of reserved keywords)
length ::= *unsigned-integer*
letter ::= *lower-case-letter* | *upper-case-letter*
like-predicate ::= *expression* [NOT] LIKE *pattern-value*
like-predicate ::=
    expression [NOT] LIKE *pattern-value* [*ODBC-like-escape-clause*]
literal ::= *character-string-literal*
literal ::= *character-string-literal* | *numeric-literal*
literal ::= *character-string-literal*
    | *numeric-literal*
    | *bit-literal*
    | *binary-literal*
    | *ODBC-date-time-extension*
lower-case-letter ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
mantissa ::= *exact-numeric-literal*
minutes-value ::= *digit digit*
months-value ::= *digit digit*
null-predicate ::= *column-name* IS [NOT] NULL
numeric-literal ::= *exact-numeric-literal* | *approximate-numeric-literal*
ODBC-date-literal ::=
    ODBC-std-esc-initiator d '*date-value*' *ODBC-std-esc-terminator*
    | *ODBC-ext-esc-initiator* d '*date-value*' *ODBC-ext-esc-terminator*
ODBC-date-time-extension ::=
    ODBC-date-literal
    | *ODBC-time-literal*
    | *ODBC-timestamp-literal*
ODBC-like-escape-clause ::=
    ODBC-std-esc-initiator escape '*escape-character*'
        ODBC-std-esc-terminator
    | *ODBC-ext-esc-initiator* escape '*escape-character*'
        ODBC-ext-esc-terminator

1001/006

ODBC-time-literal ::=
ODBC-std-esc-initiator t *'time-value' ODBC-std-esc-terminator*
| *ODBC-ext-esc-initiator* t *'time-value' ODBC-ext-esc-terminator*
ODBC-timestamp-literal ::=
ODBC-std-esc-initiator ts *'timestamp-value' ODBC-std-esc-terminator*
| *ODBC-ext-esc-initiator* ts *'timestamp-value' ODBC-ext-esc-terminator*
ODBC-ext-esc-initiator ::= {
ODBC-ext-esc-terminator ::= }
ODBC-outer-join-extension ::=
ODBC-std-esc-initiator oj *outer-join ODBC-std-esc-terminator*
| *ODBC-ext-esc-initiator* oj *outer-join ODBC-ext-esc-terminator*
ODBC-scalar-function-extension ::=
ODBC-std-esc-initiator fn *scalar-function ODBC-std-esc-terminator*
| *ODBC-ext-esc-initiator* fn *scalar-function ODBC-ext-esc-terminator*
ODBC-std-esc-initiator ::= *ODBC-std-esc-prefix SQL-esc-vendor-clause*
ODBC-std-esc-prefix ::= --(*
ODBC-std-esc-terminator ::= *)--
order-by-clause ::= ORDER BY *sort-specification* [, *sort-specification*]...
outer-join ::= *table-name* [*correlation-name*] {LEFT | RIGHT | FULL}
OUTER JOIN{*table-name* [*correlation-name*] | *outer-join*} ON *search-condition*
owner-name ::= **QP-current-qualifier**
pattern-value ::= *character-string-literal* | *dynamic-parameter*
pattern-value ::= *character-string-literal* | *dynamic-parameter* | USER
precision ::= *unsigned-integer*
predicate ::= *comparison-predicate* | *like-predicate* | *null-predicate*
predicate ::=
between-predicate | *comparison-predicate* | *exists-predicate*
| *in-predicate* | *like-predicate* | *null-predicate* | *quantified-predicate*
primary ::= *column-name*
| *dynamic-parameter*
| *literal*
| (*expression*)
primary ::= *column-name*
| *dynamic-parameter*
| *literal*
| *set-function-reference*
| USER
| (*expression*)
primary ::= *column-name*
| *dynamic-parameter*
| *literal*
| *ODBC-scalar-function-extension*
| *set-function-reference*
| USER
| (*expression*)

19

1001/006

procedure ::= *procedure-name* | *procedure-name* (*procedure-parameter-list*)
procedure-identifier ::= ~~*QP-procedure-identifier*~~
procedure-name ::= *procedure-identifier*
| *owner-name.procedure-identifier*
| *qualifier-name qualifier-separator procedure-identifier*
| *qualifier-name qualifier-separator* [*owner-name*].*procedure-identifier*
procedure-parameter-list ::= *procedure-parameter*
| *procedure-parameter, procedure-parameter-list*
procedure-parameter ::= *dynamic-parameter* | *literal* | *empty-string*
ref-table-name ::= *base-table-identifier*
qualifier-name ::= ~~*QP-current-qualifier*~~
qualifier-separator ::= {implementation-defined}
quantified-predicate ::= *expression comparison-operator* {ALL | ANY}
(*sub-query*)
query-specification ::=
SELECT [ALL | DISTINCT] *select-list*
FROM *table-reference-list*
[WHERE *search-condition*]
[GROUP BY *column-name*, [*column-name*]...]
[HAVING *search-condition*]
ref-table-name ::= *base-table-identifier*
| *owner-name.base-table-identifier*
| *qualifier-name qualifier-separator base-table-identifier*
| *qualifier-name qualifier-separator* [*owner-name*].*base-table-identifier*
referenced-columns ::= (*column-identifier* [, *column-identifier*]...)
referencing-columns ::= (*column-identifier* [, *column-identifier*]...)
scalar-function ::= *function-name* (*argument-list*)
scale ::= *unsigned-integer*
search-condition ::= *boolean-term* [OR *search-condition*]
seconds-fraction ::= *unsigned-integer*
seconds-value ::= *digit digit*
select-list ::= * | *select-sublist* [, *select-sublist*
select-sublist ::= *expression*
select-sublist ::= *expression* [[AS] *column-alias*]
| {*table-name* | *correlation-name*}.*
set-function-reference ::= COUNT(*) | *distinct-function* | *all-function*
sort-specification ::= {*unsigned-integer* | *column-name* } [ASC | DESC]
SQL-esc-vendor-clause ::= VENDOR(Microsoft), PRODUCT(ODBC)
sub-query ::=
SELECT [ALL | DISTINCT] *select-list*
FROM *table-reference-list*
[WHERE *search-condition*]
[GROUP BY *column-name* [, *column-name*]...]
[HAVING *search-condition*]
table-identifier ::= ~~*QP-base-table-name*~~
table-name ::= *table-identifier*

1001/006

```
        | owner-name.table-identifier
        | qualifier-name qualifier-separator table-identifier
        | qualifier-name qualifier-separator [owner-name].table-identifier
table-reference ::= table-name
table-reference ::= table-name [correlation-name]
table-reference ::= table-name [correlation-name]
        | ODBC-outer-join-extension
table-reference-list ::= table-reference [,table-reference]...
term ::= factor | term {*|/} factor
time-separator ::= :
time-type ::= {time types}
time-value ::=
        hours-value time-separator minutes-value time-separator
            seconds-value
timestamp-separator ::=
        (The blank character.)
timestamp-type ::= {timestamp types}
timestamp-value ::= date-value timestamp-separator
            time-value[.seconds-fraction]
timestamp-value ::= QP-sql-date-time-list
unsigned-integer ::= {digit}...
upper-case-letter ::=
        A | B | C | D | E | F | G | H | I | J | K | L | M |
        N | O | P | Q | R | S | T | U | V | W | X | Y | Z
user-defined-name ::= letter[digit | letter | _]...
user-name ::= user-defined-name
value ::= literal | USER | dynamic-parameter
viewed-table-identifier ::= user-defined-name
viewed-table-name ::= viewed-table-identifier
        | owner-name.viewed-table-identifier
        | qualifier-name qualifier-separator viewed-table-identifier
        | qualifier-name qualifier-separator [owner-name].viewed-table-identifier
years-value ::= digit digit digit digit

function-name ::= ODBC-string-functions |
                  ODBC-numeric-functions |
                  ODBC-time-and-date-functions |
                  ODBC-system-functions |
                  ODBC-convert-function

ODBC-string-functions ::= ASCII (string-exp) |
                  CHAR(code) |
                  CONCAT(string-exp1,string-exp2) |
                  DIFFERENCE(string_exp1,string_exp2) |
                  INSERT(string_exp1,start,length,string_exp2) |
                  LCASE(string_exp) |
```

1001/006

LEFT(*string_exp ,count*) |
LENGTH(*string_exp*) |
LOCATE(*string_exp1, string_exp2[,start]*) |
LTRIM(*string_exp*) |
REPEAT(*string_exp, count*) |
REPLACE(*string_exp1, string_exp2, string_exp3*) |
RIGHT(*string_exp ,count*) |
RTRIM(*string_exp*) |
SOUNDEX(*string_exp*) |
SOUNDEX(*count*) |
RIGHT(*string_exp ,start, length*) |
RTRIM(*string_exp*)

string_exp ::= ***QP-column-identifier*** |
QP-sql-char-list |
string-literal |
ODBC-string-functions
string_exp1 ::= string_exp
string_exp2 ::= string_exp
length ::= ***QP-data-element*** |
QP-sql-numeric-list |
number |
ODBC-numeric-functions
start ::= ***QP-data-element*** |
number |
QP-sql-numeric-list |
ODBC-numeric-functions
count ::= ***QP-data-element*** |
number |
QP-sql-numeric-list |
ODBC-numeric-functions

ODBC-numeric-functions ::= ABS(*numeric_exp*) |
ACOS(*float_exp*) |
ASIN(*float_exp*) |
ATAN(*float_exp*) |
ATAN2(*float_exp1, float_exp2*) |
CEILING(*numeric_exp*) |
COS(*float_exp*) |
COT(*float_exp*) |
DEGREES(*numeric_exp*) |
EXP(*float_exp*) |
FLOOR(*numeric_exp*) |
LOG(*float_exp*) |
LOG10(*float_exp*) |
MOD(*integer_exp1, integer_exp2*) |

1001/006

```
            PI() |
            POWER(numeric_exp, integer_exp) |
            RADIANS(numeric_exp) |
            RAND([integer_exp]) |
            ROUND(numeric_exp, integer_exp) |
            SIGN(numeric_exp) |
            SIN(float_exp) |
            SQRT(float_exp) |
            TAN(float_exp) |
            TRUNCATE(numeric_exp, integer_exp)

numeric_exp ::= QP-data-element |
  QP-column-identifier |
  QP-sql-numeric-list |
      number |
      ODBC-numeric-functions

float_exp ::= QP-data-element |
  QP-column-identifier |
  QP-sql-numeric-list |
      number |
      ODBC-numeric-functions

integer_exp ::= QP-data-element |
  QP-column-identifier |
  QP-sql-numeric-list |
      number |
      ODBC-numeric-functions

ODBC-time-and-date-functions ::= CURDATE() |
      CURTIME() |
      DATETIME(date_exp) |
      DAYOFMONTH(date_exp) |
      DAYOFWEEK(date_exp) |
      DAYOFYEAR(date_exp) |
      HOUR(date_exp) |
      MINUTE(date_exp) |
      MONTH(date_exp) |
      MONTHNAME(date_exp) |
      NOW() |
      QUARTER(date_exp) |
      SECOND(date_exp) |
      TIMESTAMPADD(interval, integer_exp, timestamp_exp) |
      TIMESTAMPDIFF(interval, timestamp_exp1 , timestamp_exp2) |
      WEEK(date_exp) |
      YEAR(date_exp)
```

```
1001/006

date_exp ::= QP-data-element |
  QP-column-identifier |
  QP-sql-date-time-list |
      number |
      ODBC-time-and-date-functions

timestamp_exp ::= QP-data-element |
  QP-column-identifier |
  QP-sql-date-time-list |
      ODBC-time-and-date-functions

interval ::= SQL_TSI_FRAC_SECOND |
     SQL_TSI_SECOND |
     SQL_TSI_MINUTE |
     SQL_TSI_HOUR |
     SQL_TSI_DAY |
     SQL_TSI_WEEK |
     SQL_TSI_MONTH |
     SQL_TSI_QUARTER |
     SQL_TSI_YEAR |

ODBC-system-functions ::= DATABASE() |
              IFNULL(exp, value) |
              USER() |

exp ::= column-name
exp ::= column-name QP-math-operation-list column-name
value ::= QP-data-element

ODBC-convert-function ::= CONVERT(QP-column-identifier , QP-sql-data-type-list-element)
ODBC-convert-function ::= CONVERT(value-exp, data-type)

ODBC-data-type ::= SQL_CHAR |
              SQL_VARCHAR |
              SQL_LONGVARCHAR |
              SQL_DECIMAL |
              SQL_NUMERIC |
              SQL_SMALLINT |
              SQL_INTEGER |
              SQL_REAL |
              SQL_FLOAT |
              SQL_DOUBLE |
              SQL_TINYINT |
              SQL_BIGINT |
     SQL_BINARY |
     SQL_VARBINARY |
```

1001/006

```
    SQL_LONGVARBINARY |
    SQL_DATE |
    SQL_TIMESTAMP

ODBC-char-type ::= SQL_CHAR |
            SQL_VARCHAR |
            SQL_LONGVARCHAR

ODBC-numeric-type ::= SQL_DECIMAL |
            SQL_NUMERIC |
            SQL_SMALLINT |
            SQL_INTEGER |
            SQL_REAL |
            SQL_FLOAT |
            SQL_DOUBLE |
            SQL_TINYINT |
            SQL_BIGINT

ODBC-binary-type ::= SQL_BINARY |
    SQL_VARBINARY |
    SQL_LONGVARBINARY

ODBC-date-time-type ::= SQL_DATE |
    SQL_TIMESTAMP

QP-function-name ::=      ASCII()
QP-procedure-identifier ::=      <index-qualifier QP-number>
QP-index-qualifier ::=      <index-qualifier QP-number>
QP-cursor-name ::=      <cursor QP-number>
QP-index-name ::= <create table QP-number>
QP-current-qualifier ::= <qualifier>
QP-base-table-name ::= <table QP-number>
QP-table-extension ::= <ext>
QP-column-identifier ::= <column QP-number>
QP-alias ::= <alias number>
QP-data-element ::= <data QP-number >
QP-column-name ::= <column name>
QP-column-definition ::= <column def>
QP-list ::= [QP-comparison-list |
                QP-outer-join-list |
                QP-math-operation-list |
                QP-sql-data-type-list |
        QP-sql-date-time-list |
        QP-sql-numeric-list |
        QP-sql-char-list |
        QP-clause-list ]
```

25

1001/006

QP-clause-list ::= *QP-clause-list-element*

QP-clause-list-element ::= *QP-clause-list-element* |
HAVING |
GROUP-BY |
ORDER-BY
QP-comparison-list ::= *QP-comparison-list-element*
QP-outer-join-list ::= *QP-outer-join-list-element*
QP-math-operation-list ::= *QP-math-operation-list-element*
QP-sql-data-type-list ::= *QP-sql-data-type-list-element*
QP-number ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10...
QP-comparison-list-element ::= *comparison-list-element* |
= | < | <= | > | >= | != | !< | !
QP-outer-join-list-element ::= *outer-join-list-element* |
= | =
QP-math-operation-element ::= *QP-math-operation-element* |
- | + | * | / | %
QP-sql-data-type-list-element ::= *QP-sql-data-type-list-element* |
ODBC-data-type

QP-sql-date-time-list ::= *ODBC-date-time-type*
QP-sql-numeric-list ::= *ODBC-numeric-type*
QP-sql-char-list ::= *ODBC-char-type*
QP-sql-binary-list ::= *ODBC-binary-type*

SQL Statements:

statement := *alter-table-statement* |
create-index-statement |
create-table-statement |
create-view-statement |
delete-statement-positioned |
delete-statement-searched |
drop-index-statement |
drop-table-statement |
drop-view-statement |
grant-statement |
insert-statement |
revoke-statement |
select-statement |
select-for-update-statement |
update-statement-positioned |
update-statement-searched |

alter-table-statement ::=
ALTER TABLE *base-table-name*

1001/006

```
        {   ADD column-identifier data-type
        |   ADD (column-identifier data-type [, column-identifier data-type]... )
        }

alter-table-statement ::=
        ALTER TABLE base-table-name
        {   ADD column-identifier data-type
        |   ADD (column-identifier data-type [, column-identifier data-type]... )
        |   DROP [COLUMN] column-identifier [CASCADE | RESTRICT]
        }

create-index-statement ::=
        CREATE [UNIQUE] INDEX index-name
        ON base-table-name
        ( column-identifier [ASC | DESC]
        [, column-identifier [ASC | DESC] ]... )

create-table-statement ::=
        CREATE TABLE base-table-name
        (column-element [, column-element] ...)
        column-element ::= column-definition | table-constraint-definition
        column-definition ::=
                column-identifier data-type
                [DEFAULT default-value]
                [column-constraint-definition[ column-constraint-definition]...]

column-constraint-definition ::=
                NOT NULL
                | UNIQUE | PRIMARY KEY
                | REFERENCES ref-table-name referenced-columns
                | CHECK (search-condition)

table-constraint-definition ::=
                UNIQUE (column-identifier [, column-identifier] ...)
                | PRIMARY KEY (column-identifier
                    [, column-identifier] ...)
                | CHECK (search-condition)
                | FOREIGN KEY referencing-columns REFERENCES
                    ref-table-name referenced-columns

create-view-statement ::=
        CREATE VIEW viewed-table-name
        [( column-identifier [, column-identifier]... )]
        AS query-specification

delete-statement-positioned ::=
        DELETE FROM table-name WHERE CURRENT OF cursor-name
```

1001/006

delete-statement-searched ::=
DELETE FROM *table-name* [WHERE *search-condition*]

drop-index-statement ::=
DROP INDEX *index-name*

drop-table-statement ::=
DROP TABLE *base-table-name*
[CASCADE | RESTRICT]

drop-view-statement ::=
DROP VIEW *viewed-table-name*
[CASCADE | RESTRICT]

grant-statement ::=
GRANT {ALL | *grant-privilege* [, *grant-privilege*]... }
ON *table-name*
TO {PUBLIC | *user-name* [, *user-name*]... }
grant-privilege ::=
DELETE
| INSERT
| SELECT
| UPDATE [(*column-identifier* [, *column-identifier*]...)]
| REFERENCES [(*column-identifier*
[, *column-identifier*]...)]

insert-statement ::=
INSERT INTO *table-name* [(*column-identifier* [, *column-identifier*]...)]
VALUES (*insert-value*[, *insert-value*]...)

insert-statement ::=
INSERT INTO *table-name* [(*column-identifier* [, *column-identifier*]...)]
{ *query-specification* | VALUES (*insert-value* [, *insert-value*]...)}

revoke-statement ::=
REVOKE {ALL | *revoke-privilege* [, *revoke-privilege*]... }
ON *table-name*
FROM {PUBLIC | *user-name* [, *user-name*]... }
[CASCADE | RESTRICT]
revoke-privilege ::=
DELETE
| INSERT
| SELECT
| UPDATE
| REFERENCES

1001/006

select-statement ::=
SELECT [ALL | DISTINCT] *select-list*
FROM *table-reference-list*
[WHERE *search-condition*]
[*order-by-clause*]

select-statement ::=
SELECT [ALL | DISTINCT] *select-list*
FROM *table-reference-list*
[WHERE *search-condition*]
[GROUP BY *column-name* [, *column-name*]...]
[HAVING *search-condition*]
[*order-by-clause*]

select-statement ::=
SELECT [ALL | DISTINCT] *select-list*
FROM *table-reference-list*
[WHERE search-condition]
[GROUP BY *column-name* [, *column-name*]...]
[HAVING *search-condition*]
[UNION [ALL] *select-statement*]...
[*order-by-clause*]

select-for-update-statement ::=
SELECT [ALL | DISTINCT] *select-list*
FROM *table-reference-list*
[WHERE *search-condition*]
FOR UPDATE OF [*column-name* [, *column-name*]...]

update-statement-positioned ::=
UPDATE *table-name*
SET *column-identifier* = {*expression* | NULL}
[, *column-identifier* = {*expression* | NULL}]...
WHERE CURRENT OF *cursor-name*

update-statement-searched
UPDATE *table-name*
SET *column-identifier* = {*expression* | NULL }
[, *column-identifier* = {*expression* | NULL}]...
[WHERE *search-condition*]

While the invention has been illustrated and described in detail in the drawings and foregoing description, such illustration and description is to be considered as exemplary and not restrictive in character, it being understood that only the preferred embodiment and minor variants thereof have been shown and described and that all changes and modifications that come within the spirit of the invention are desired to be protected.

What is claimed is:

1. Computer interpretable grammatical rules for generating a plurality of queries in a grammar for testing a database engine driver, said grammatical rules comprising:

static elements for generating constant portions of said plurality of queries, wherein said static elements are copied to a buffer associated with a computer to generate said plurality of queries from said grammatical rules; and

variable elements selected from at least one of a group consisting of: a query element and a query list element, for generating database engine driver specific portions of said plurality of queries, wherein said variable elements are replaced in said buffer by values specific to a particular database engine driver to generate said plurality of queries from said, grammatical rules.

2. The grammatical rules of claim 1 wherein said query element is enclosed by a start delimiter and an end delimiter.

3. The grammatical rules of claim 2 wherein said start delimiter is a less-than character ("<") and said end delimiter is a greater-than character (">"), and a comma character (",") separates adjacent ones of said plurality of values in said query element.

4. The grammatical rules of claim 1 wherein said query list element is enclosed by a start delimiter and an end delimiter.

5. The grammatical rules of claim 4 wherein said start delimiter is a left square brace character ("["), and said end delimiter is a right square brace character ("]"), and a comma character (",") separates adjacent ones of said plurality of values in said query list element.

6. A computer operable method for testing a database driver, said method comprising:

parsing a meta-language statement into at least one meta-language statement token each comprised of at least one token element where any one of said at least one meta-language statement token that is comprised of more than one token element is a variable token element delimited by a pair of variable token element delimiters and said variable token element is a type selected from at least one of a group consisting of: a query element and a query list element;

expanding said meta-language statement into a plurality of meta-language test queries comprised of one of said plurality of meta-language test queries for each unique combination of said at least one token element in each of said at least one meta-language statement token;

generating a plurality of data type specific test queries from said plurality of meta-language test queries by direct substitution of a data type specific database driver query element for each substitutable one of said at least one token element in each of said plurality of meta-language test queries;

repeating said step of generating for each data type supported by said database driver; and

applying said plurality of data type specific test queries to said database driver.

7. A method according to claim 6 wherein said pair of variable token element delimiters for said query element include a less-than symbol ("<") as a start delimiter and a greater-than symbol (">") as an end delimiter, and individual tokens of said query element are separated by a comma symbol (",").

8. A method according to claim 6 wherein said pair of variable token element delimiters for said query list element include a left square brace symbol ("[") as a start delimiter and a right square brace symbol ("]") as an end delimiter, and individual tokens of said query list element are separated by a comma symbol (",").

9. A rule-based test apparatus for generating a plurality of test commands to test a database engine driver, said apparatus comprising:

a memory;

a processor connected to said memory;

a plurality of rules, stored in said memory, wherein each of said plurality of rules is encoded in a meta-language syntax used to represent a plurality of test commands that are executable on corresponding ones of a plurality of database engine drivers; and

processing means, operable in said processor, for parsing a meta-language statement input and for generating each of said plurality of test commands according to said plurality of rules and for applying each of said plurality of test commands to said database engine driver.

* * * * *

US006363391B1

(12) **United States Patent**
Rosensteel, Jr.

(10) **Patent No.: US 6,363,391 B1**
(45) **Date of Patent: Mar. 26, 2002**

(54) **APPLICATION PROGRAMMING INTERFACE FOR MONITORING DATA WAREHOUSE ACTIVITY OCCURRING THROUGH A CLIENT/SERVER OPEN DATABASE CONNECTIVITY INTERFACE**

(75) Inventor: **Kenneth R. Rosensteel, Jr.**, Phoenix, AZ (US)

(73) Assignee: **Bull HN Information Systems Inc.**, Billerica, MA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/086,818**

(22) Filed: **May 29, 1998**

(51) **Int. Cl.**[7] **G06F 17/30**
(52) **U.S. Cl.** **707/102**; 709/223; 707/2; 707/10
(58) **Field of Search** 707/10, 2, 100–103; 709/223

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,675,785	A *	10/1997	Hall et al.	707/102
5,721,903	A *	2/1998	Anand et al.	707/5
5,832,496	A *	11/1998	Anand et al.	707/102
5,870,746	A *	2/1999	Knutson et al.	707/101
5,918,224	A *	6/1999	Bredenberg	707/2
5,918,225	A *	6/1999	White et al.	707/3
5,970,490	A *	10/1999	Morgensten	707/10
6,167,405	A *	12/2000	Rosensteel, Jr. et al.	707/102

OTHER PUBLICATIONS

Data Warehousing An Introduction, by Grayce Booth, Groupe Bull Technical Update, May/Jun. 1995, pp. 1–9, Copyright 6/95, Bull S.A., France.

The Distributed Data Warehouse Solution, by Kirk Mosher & Ken Rosensteel, Groupe Bull Technical Update, May/Jun. 1995, pp. 11–18, Copyright 6/95, Bull S.A., France.

System Security and Management in a Distributed Data Warehouse Environment, by Denis Pinkas, Ken Rosensteel, and Martin Schiavo, Groupe Bull Technical Update, vol. 6, No. 2, pp. 13–22, Copyright.

Bull HN Information Systems Inc. and Bull S.A., 1996, Billerica, MA. and France.

DDW Adminstrator's Guide, Order No. 86 A2 83FC REV 4, Release Date: Apr. 25, 1997, Copyright Bull S.A & Bull HN 1995, 1996, 1997.

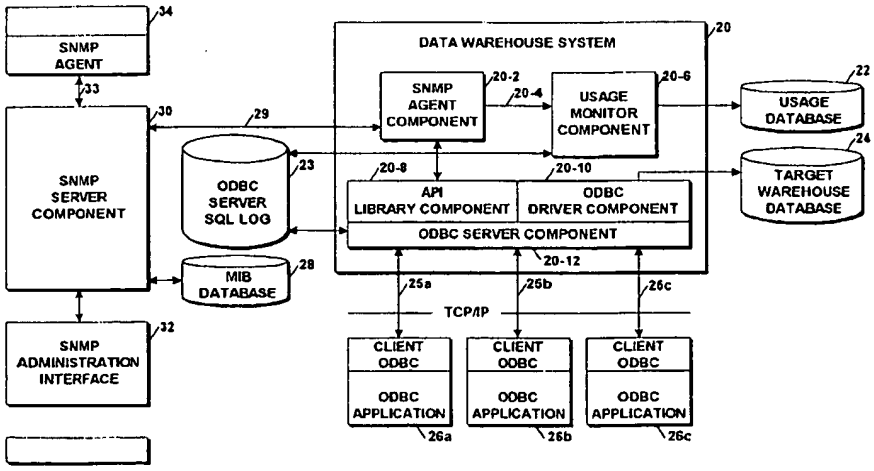
* cited by examiner

Primary Examiner—Hosain T. Alam
Assistant Examiner—Jean Bolte Fleurantin
(74) *Attorney, Agent, or Firm*—Faith F. Driscoll; John S. Solakian

(57) **ABSTRACT**

An Application Programming Interface (API) provides interoperability between different monitoring and administrative components of a data warehouse system that utilizes different standard protocols. One of the protocols is the well known data connectivity protocol, Open Database Connectivity (ODBC) that defines a standard interface between applications and data sources. A second one of the protocols is the well known network management protocol, Simple Network Management Protocol (SNMP) that defines a standard interface between an agent component and a network management system. The API provides a facility that enables the different components to access user and connection information maintained by an ODBC server component derived from servicing client system application SQL queries made by system users.

34 Claims, 3 Drawing Sheets



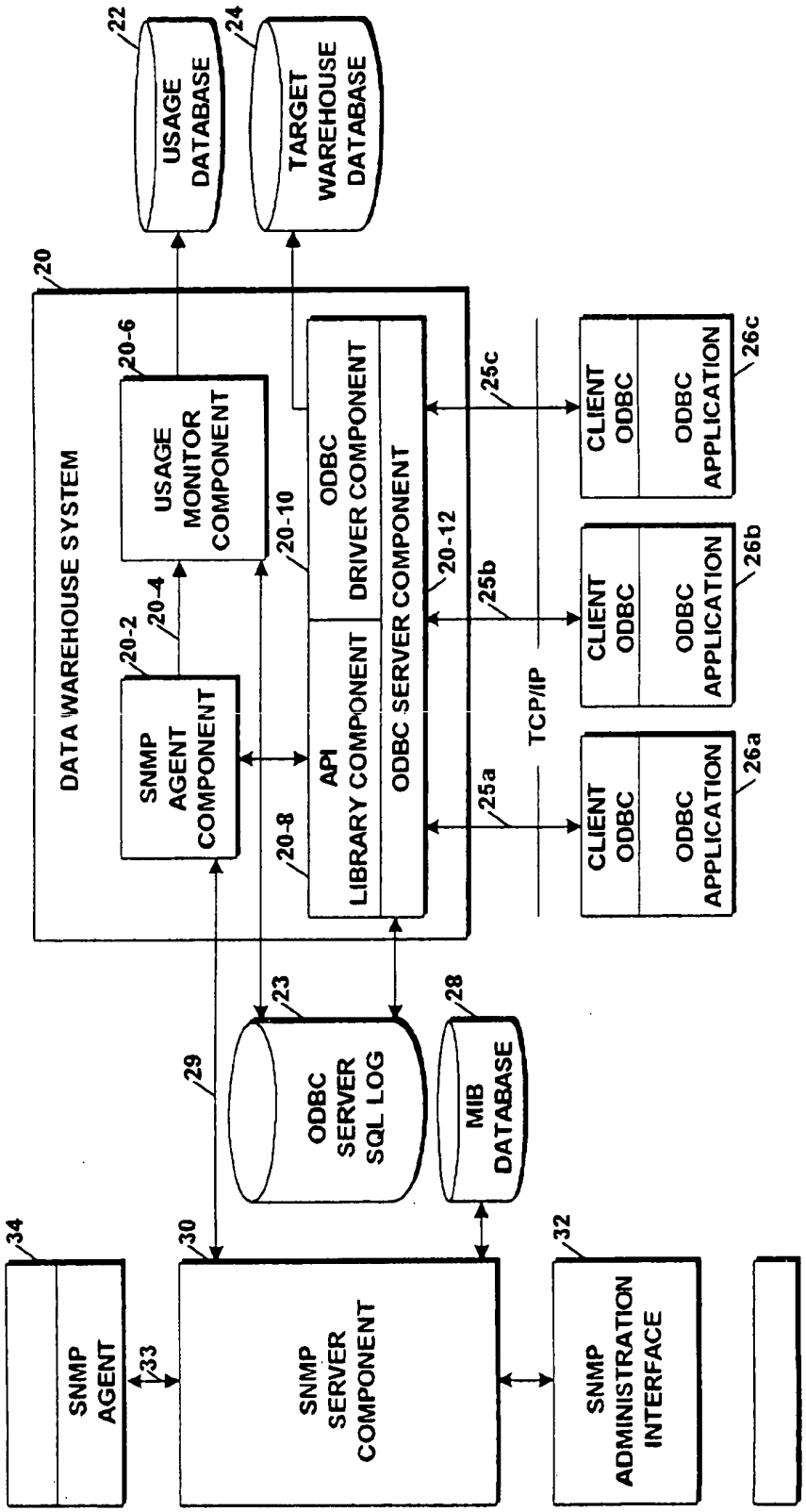

22
USAGE DATABASE
24
TARGET WAREHOUSE DATABASE
20
DATA WAREHOUSE SYSTEM
20-6
USAGE MONITOR COMPONENT
20-10
ODBC DRIVER COMPONENT
20-12
ODBC SERVER COMPONENT
20-2
20-4
SNMP AGENT COMPONENT
20-8
API LIBRARY COMPONENT
25c
25b
25a
TCP/IP
CLIENT ODBC
ODBC APPLICATION
26c
26b
26a
23
ODBC SERVER SQL LOG
28
MIB DATABASE
29
34
SNMP AGENT
33
30
SNMP SERVER COMPONENT
32
SNMP ADMINISTRATION INTERFACE
10

FIGURE 1

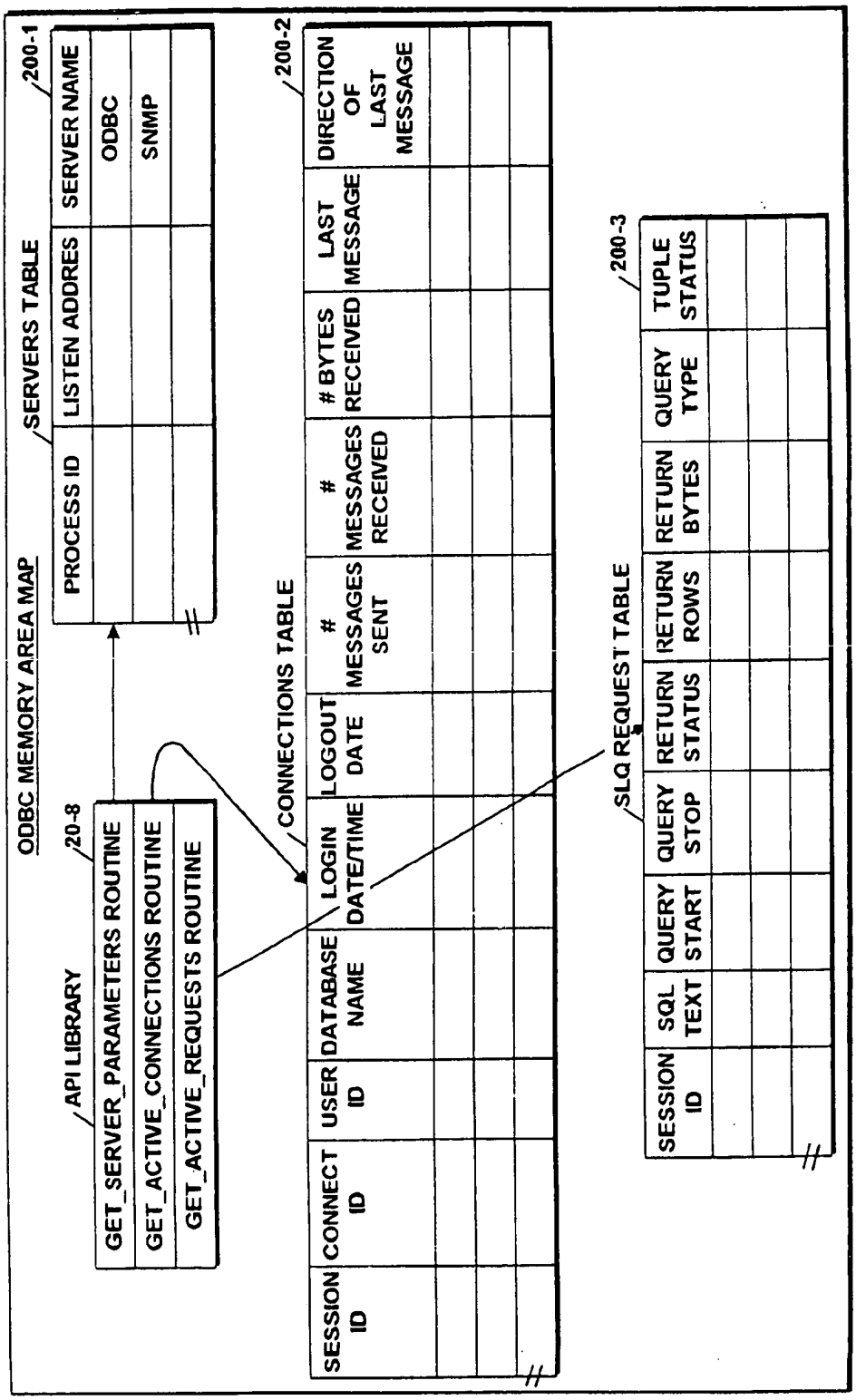

ODBC MEMORY AREA MAP
API LIBRARY
20-8
GET_SERVER_PARAMETERS ROUTINE
GET_ACTIVE_CONNECTIONS ROUTINE
GET_ACTIVE_REQUESTS ROUTINE
SERVERS TABLE
200-1
PROCESS ID
LISTEN ADDRES
SERVER NAME
ODBC
SNMP
CONNECTIONS TABLE
200-2
SESSION ID
CONNECT ID
USER ID
DATABASE NAME
LOGIN DATE/TIME
LOGOUT DATE
MESSAGES SENT
MESSAGES RECEIVED
BYTES RECEIVED
LAST MESSAGE
DIRECTION OF LAST MESSAGE
SLQ REQUEST TABLE
200-3
SESSION ID
SQL TEXT
QUERY START
QUERY STOP
RETURN STATUS
RETURN ROWS
RETURN BYTES
QUERY TYPE
TUPLE STATUS

FIGURE 2

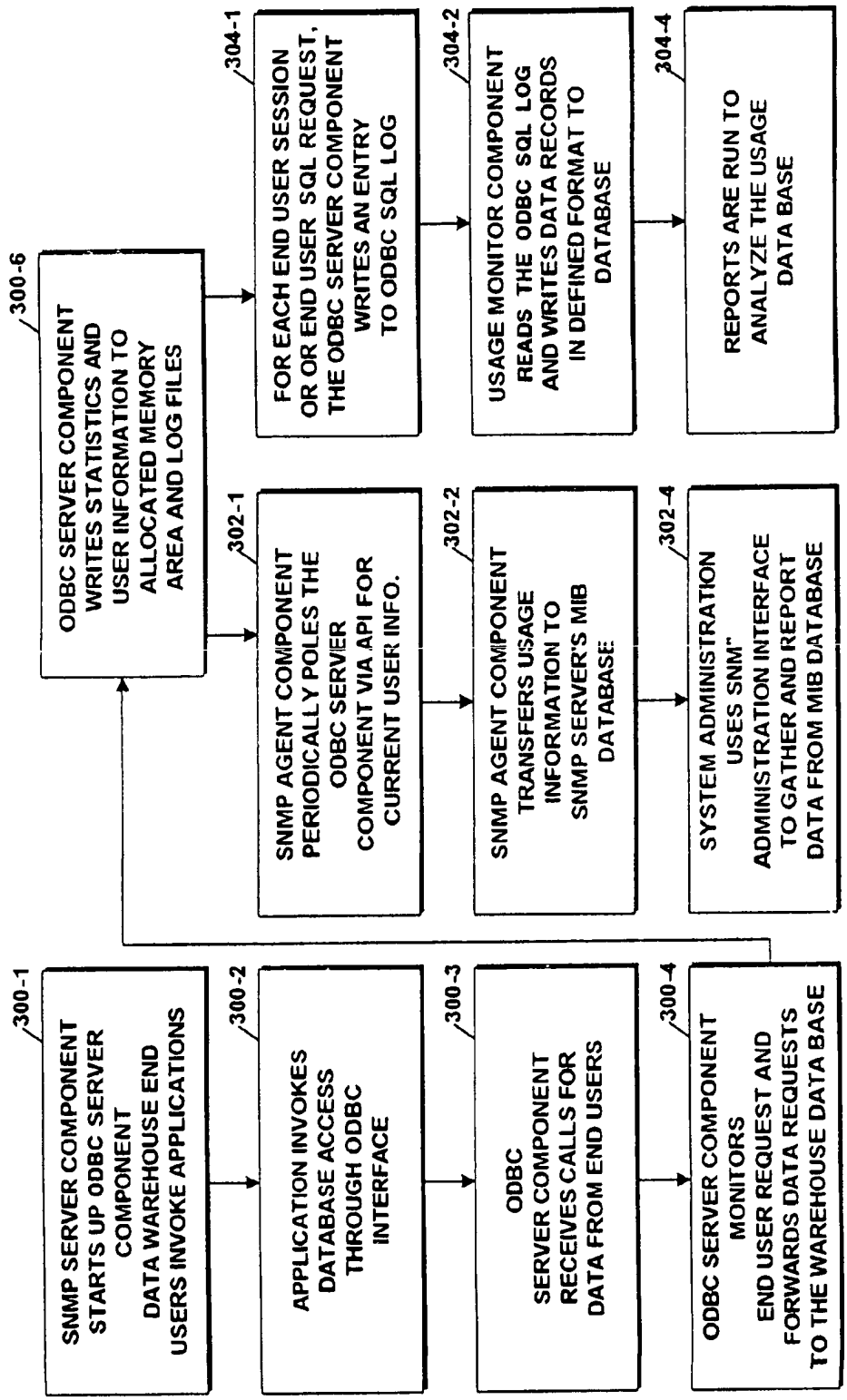

300-1
SNMP SERVER COMPONENT STARTS UP ODBC SERVER COMPONENT DATA WAREHOUSE END USERS INVOKE APPLICATIONS
300-2
APPLICATION INVOKES DATABASE ACCESS THROUGH ODBC INTERFACE
300-3
ODBC SERVER COMPONENT RECEIVES CALLS FOR DATA FROM END USERS
300-4
ODBC SERVER COMPONENT MONITORS END USER REQUEST AND FORWARDS DATA REQUESTS TO THE WAREHOUSE DATA BASE
300-6
ODBC SERVER COMPONENT WRITES STATISTICS AND USER INFORMATION TO ALLOCATED MEMORY AREA AND LOG FILES
302-1
SNMP AGENT COMPONENT PERIODICALLY POLES THE ODBC SERVER COMPONENT VIA API FOR CURRENT USER INFO.
302-2
SNMP AGENT COMPONENT TRANSFERS USAGE INFORMATION TO SNMP SERVER'S MIB DATABASE
302-4
SYSTEM ADMINISTRATION USES SNM" ADMINISTRATION INTERFACE TO GATHER AND REPORT DATA FROM MIB DATABASE
304-1
FOR EACH END USER SESSION OR OR END USER SQL REQUEST, THE ODBC SERVER COMPONENT WRITES AN ENTRY TO ODBC SQL LOG
304-2
USAGE MONITOR COMPONENT READS THE ODBC SQL LOG AND WRITES DATA RECORDS IN DEFINED FORMAT TO DATABASE
304-4
REPORTS ARE RUN TO ANALYZE THE USAGE DATA BASE

FIGURE 3

APPLICATION PROGRAMMING INTERFACE FOR MONITORING DATA WAREHOUSE ACTIVITY OCCURRING THROUGH A CLIENT/SERVER OPEN DATABASE CONNECTIVITY INTERFACE

BACKGROUND OF THE INVENTION

1. Field of Use

The present invention relates to systems and methods for monitoring information accesses and more particullrly the usage of a data warehouse and the information contained therein.

2. Prior Art

Data warehouses are becoming more and more important to businesses. The term "data warehouse" is generally used to describe a database containing data that was gathered from a variety of sources (e.g. existing production databases). For more information regarding the nature of a data warehouse, reference may be made to the article entitled, "Data Warehousing: An Introduction" by Grayce Booth which appeared in the May/June 1995 issue of the Bull S. A. technical journal entitled, "Technical Update."

Typically, the data warehouse is implemented as a large amount of data stored in a database with access to the data coming from hundreds of users executing commodity applications like Excel, running on personal computers (PCs). Here, an opportunity for a business exists to manage the data warehouse system. It is useful to the warehouse owner to have information and statistics about the usage of the warehouse and its data. Such information includes: (a) how many users are currently logged onto the system; (b) what is the pattern of access statistics; (c) what data is accessed most frequently, (d) what if any indexes could be added or dropped to improve access efficiency; (e) what if any unlawful access attempts have occurred; and (f) what query runs the longest. Some of this information can be obtained from the warehouse database system but each type of database gathers this information in a different proprietary manner. Therefore, there is an opportunity to be able to provide usage data in a standard fashion for all database types. Also, there is the ability to provide the information through standard system management tools based on standard protocols, such as the Simple Network Management Protocol (SNMP).

As well known in the art, the Open Data Base Connectivity (ODBC) application programming interface is a standard defined by Microsoft Corporation by which Windows based tools and applications may access different databases on many different server platforms. Many PC vendors have adopted ODBC. Without using ODBC, applications are required to use APIs specific to a database vendor for accessing data warehouse information. Using ODBC, an application may access any type of database. In addition, ODBC is used by application tools such as EXCEL such that specific code is not required for each database type being accessed.

Client/Server ODBC is a newer technique for implementing ODBC. The interface to ODBC for user applications remains on the PC but the bulk of the ODBC logic is moved to a server side implementation. All PC users execute their data requests through a common ODBC server. This arrangement provides a "thin" client requirement for the PC user of ODBC and makes the administration of ODBC possible from a single server. This single point of access through the ODBC server also provides the opportunity for administrating the data warehouse. All clients PCs that need to access the data warehouse come through the single point of access (Le. ODBC server).

Although ODBC provides a common PC based API, each relational database management system (RDBMS) vendor typically has implemented a unique interface for data access. To adapt tools based on ODBC to the interfaces used by various types of RDBMS, Microsoft Corporation specifies the development of a "driver". The driver transforms the ODBC API standard calls into RDBMS specific calls. The use of ODBC provides a layer of consistency above each of the APIs implemented by the RDBMS vendors. In the prior art, a separate ODBC driver was required for each type of RDBMS to be accessed. Additionally, each database vendor typically, requires a tailored communications link An improvement to this approach is to provide a single data access (DDA)ODBC driver to replace multiple customized ODBC drivers with a single implementation that can access multiple types of databases.

An example of the above type of system is the distributed data warehouse (DDW) middleware described in the article entitled, "The Distributed Data Warehouse Solution"by Kirk Mosher and Ken Rosensteel that also appeared in the above referenced May/June 1995 of the Technical Update Journal. This system utilizes a proprietary based infrastructure called DDW/NET that works in conjunction with the DDW/ODBC driver. DDW/NET enables connections to multiple computer architectures, operating systems, and network protocols. The DDW/NET software resides on each of the legacy and server systems that communicate over standard communications links and hides the details of networking from the upper layers of software on each system.

The above prior art system included several features to aid the administrator of the data warehouse. Such features included an SNMP agent that monitored the activity of the distributed data warehouse (DDW) processes and users of the data warehouse and a Usage Monitor facility that recorded SQL database queries issued by individual users. Each of these features required the use of an interface to the DDW Net on a UNIX based platform to help gather the required information. This approach required the use of proprietary interfaces that made it difficult to expand the types of databases used by the system. The data that was needed was not easily accessible from the DDW Net memory. DDW Net design was based on Ingres technology, that could not be easily enhanced. This prior art approach is described in the publication entitled DDW Administrator's Guide, dated Apr. 25, 1997, copyright Bull S. A. and Bull HN Information Systems Inc. 1995, 1996, 1997, Order Number 86 A2 83FC Rev4.

Accordingly, it is a primary object of the present invention to provide a system and method for facilitating monitoring of data warehouse activity.

It is a further more specific object of the present invention to provide an interface arrangement that simplifies data warehouse monitoring through standard protocols.

SUMMARY OF THE INVENTION

The above objects are achieved in a preferred embodiment of the present invention that provides a special application programming interface (API) that provides interoperability between standard protocols utilized in conjunction with the monitoring and administration managing tool components of a data warehouse system. One protocol is the well known data connectivity protocol Open Database Connectivity (ODBC), that. defines a standard interface between applications and data sources. Another protocol is the well known network management protocol Simple Network Management Protocol (SNMP) that defines a standard interface between an agent component and a network management system.

In the preferred embodiment, the warehouse components include a local SNP agent component for gathering data pertaining to the activity of a distributed data warehouse (DDW) processes and the users of the DDW system and a usage monitor component for tracking statistics about the different types of SQL queries issued by individual system users. According to the present invention, the warehouse components further include ODBC server and driver components for operatively connecting to the DDW system target warehouse database for processing SQL queries submitted by warehouse knowledge workers. The ODBC server component also operatively couples to an SQL log that it uses to maintain entries pertaining to user SQL queries it receives from a number of ODBC client user systems. The usage monitor component operatively couples to the SQL log and performs the function of gathering data from the entries that it uses to populate tables of a usage monitor database that it maintains for providing usage statistics.

The SNMP agent component performs further monitoring functions. The component operatively couples to the ODBC server component through the special API that enable such components to have access to a variety of types of information received from the ODBC server through the ODBC protocol and reportistore such information in a MIB database of a further warehouse component that corresponds to a centrally located SNMP server component via the SNMP protocoL

In accordance with the present invention, the special API provides the following types of information: Server Listen Address and Number of Active Connections. For each active connection, the Connection ID, login time, number of messages sent, number of messages received, number of bytes received, last message and last message direction. Additionally, the API provides other configuration information relevant to the server such as network ports used and server name. The special API is used by the local SNMP agent component to gather real time information about data warehouse usage and reports that information to the centrally located manager server unit.

In accordance with the teachings of the present invention, the ODBC server component operatively records entries having a predetermined format (e.g. ASCII format) into two log files. Entries for every user login to the ODBC server component are recorded in the first log file. Every SQL statement sent to the ODBC server component and information identifying the user that issued the statement, the time of execution, the elapsed time of the etc. is recorded in the second log file. The usage monitor component periodically reads the second log file and writes the statistics about usage into the usage monitor database. By using the ODBC serve information, no software components need be inserted between the end users and the data warehouse to gather the usage information. Since the OD

The above objects and advantages of the present invention will be better understood from the following description when taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is an overall block diagram of a data warehousing system that includes the API of the present invention.

FIG. **2** illustrates a memory map organized according to the present invention.

FIG. **3** is a flow chart used in describing the operation of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

FIG. 1

FIG. **1** is a block diagram of a distributed data warehousing (DDW) system **10** that includes the Application Programming Interface (API) of the present invention. As shown, the DDW system **10** includes a target data warehouse system **20** that operatively connects to a plurality of ODBC Windows based personal computer (PC) client systems **26***a* through **26***c* over a corresponding number of communications links **25***a* through **25***c* and to an SNMP server component **30** over a communications link **29**. The data warehouse system **20** manages a target warehouse database **23** representing the database that implements the data warehouse or data mart. As indicated, the DDW system **10** utilizes client/server ODBC technology that removes the requirement for a DBMS network connection on the client system and software requirements. That is, the PC software requirement is reduced to the application to be run, the ODBC client software and a WinSock TCP/IP connection. The interface to the ODBC client system remains consistent with the ODBC interface standard requiring no changes to existing applications. An ODBC server component **20-12** is used to make the connection to the client systems. The ODBC server component **20-12** acts as an application to a standard ODBC driver component such as component **20-10**. The ODBC server component **20-12** runs as any other application in an assigned area of memory that it uses to provide routines and store table structures required for processing database queries and responding to API calls. The organization of the ODBC allocated memory area will be discussed in greater detail with reference to FIG. **2**.

The ODBC driver component **20-10** provides the access to the target database **24** that may be implemented as any one of a number of well-known vendor database systems (e.g. ORACLE, INFORM=, etc.). When more than one type of database system is used, the ODBC server component **20-12** utilizes Windows ODBC driver manager software. This software provides a thin layer interface to a number of different DBMS ODBC drivers that enables applications to easily load such drivers into their applications and thus remain DBMS independent

As shown, the ODBC server component **20-12** operatively couples to an ODBC server SQL database/log **23** that it uses to maintain information entries having a predetermined format for recording client application DBMS accesses. Each database write is made via the ODBC server component **20-12** that in turn results in the appropriate log entries being written into the SQL log **23**.

For the purposes of the present invention, the ODBC components may be implemented with standard ODBC software components provided by Microsoft Corporation. The implementation of ODBC drivers is well known in the art. For example, reference may be made to an article entitled "Writing ODBC Drivers" by Dennis R. McCarthy, published in the November 1995 issue of the publication, Dr. Dobb's Journal

As shown, the SNMP server component **30** also connects to other SNMP agents via communications links such as SNMP agent **34** via a communications link **33**. More specifically, SNMP server component **30** includes SNMP dispatcher daemon software that enables the component to host more than one SNMP agent at the same time. The SNMP server **30** includes an SNMP administration interface **32** and a Management Information Base (MIB) database **28**. The MIB database **28** is organized in a tree structure whose

branches identify information objects. Each object within the MIB corresponds to an item of information In the preferred embodiment, a level of the tree structure is allocated to data warehouse objects. This organization is described in greater detail, in the above referenced DDW Administrator's Guide. An appendix included herein provides a list of data warehouse objects utilized by the API of the present invention.

The system **20** includes a set of tool components that includes an SNMP agent component **20-2** that operatively couples to a usage monitor component **20-6**. The SNMT agent component **20-2** is the facility utilized to administer the data warehouse system **20** from the SNMP manager server unit **30** via communications link **29** utilizing the SNMP protocol.

The SNMP agent component **20-2** contains the necessary mechanisms for gathering data from the ODBC server component **20-12** as described herein The SNMP agent component **20-2** utilizes the SNMP server unit's MIB database **23** for recording required information utilized in monitoring activities conducted via the SNMP administration interface **32**. In the preferred embodiment, the SNMP server unit **30** and interface **32** corresponds to the ISM server developed and marketed by Bull HN Information Systems Inc.

The usage monitor component **20-6** also operatively couples to a usage database **22** and to the ODBC server SQL log **23**. The component **20-6** uses the information contained in log **23** for generating and maintaining statistics pertaining to client user database activities.

The SNMP agent component **20-2** operatively couples to the ODBC server component **20-12** through an API library component **20-8** constructed according to the teachings of the present invention. As described herein in greater detail the API library component **20-8** that can be considered part of the ODBC server component **20-12**. The component **20-8** provides a number of functions that enable the tool components of the warehouse system **20** to cooperate with the ODBC server component **20-12** in a manner required to carry out the required monitoring and administration of client system activities. FIG. **2**-Memory Map

FIG. **2** illustrates in greater detail, the organization of the allocated memory area utilized by the ODBC server component **20-12**. As shown, the memory area includes the API library component **20-2** and several key tables utilized by ODBC server component **20-12**. These key tables are: a servers table **200-1** for keeping track of servers (ODBC or SNMP), a connections table **200-2** for keeping track of active end user connections that are using ODBC client software to query the data warehouse, and a requests table **200-3** for keeping track of active SQL requests from the end users.

The API library has one function call per each of the ODBC server tables. Each of the API library calls will return all of the rows of the table being addressed. The first invocation of the function takes a snapshot of the ODBC server memory and allows all of the ODBC operations that will affect the contents of the memory to continue. This first call then returns the first entry from the table. Subsequent calls to the API return the subsequent rows of the snapshot, until an end of file (EOF) indicator is reached. The EOF is signaled through a "RETURN_CODE" parameter of the function call. The API functions are set forth in greater detail in an Appendix.

DESCRIPTION OF OPERATION

With reference to FIGS. **1** through **3**, the operation of the preferred embodiment of the present invention will now be described relative to the flowchart of FIG. **3**. Referring to FIG. **3**, it is seen from block **300-1** that initially, the systems are activated. That is, the ODBC server component **20-12** is started up by the administration SNMP Administration interface component **32**. Startup is invoked using appropriately Korn shell scripts implemented in a conventional manner. In addition, ODBC applications such as EXCEL are assumed to have been loaded and running on the client systems **26***a* through **26***c*. During the running of these applications, end users issue queries that invoke access to target warehouse database **24** as indicated by block **300-2**. This results in the ODBC client software generating calls to the ODBC server component **20-12** as indicated in block **300-3**. Each call utilizes the ODBC interface protocol that proceeds via TCP/IP network interfaces.

As indicated in block **300-4**, the ODBC server component **20-12** monitors for end user requests and forwards the end user SQL access request to the data warehouse database **24** via ODBC driver component **20-10**. In response to each call, as indicated in block **300-6**, the ODBC server component **20-12** stores the pertinent statistics and user/connection information as entries in the tables **200-1** through **200-3** located in its memory area as indicated in FIG. **2**. As discussed herein, it also stores information entries pertaining to the particular SQL query obtained from its tables in the ODBC SQL log **23**. Such entries are recorded in two ASCII log files. The first log receives an entry for every user login to the ODBC server component **20-12**, the entry is designated as a user session record that contains a number of specified attributes. The second log file receives the SQL statement received by the ODBC server component **20-12**, as well as a number of attributes such as information indicating the user that issued the statement, the time, etc.

In greater detail the ODBC SQL log files are formatted to contain the following information:

Log 1: User_Session record (attributes-User_name, database-name, login-date, logout_date, Session ID);

Log 2: User_Requests record (attributes-Session_ID, SQL_Text, Query-time, Return-Status, Return_Rows, Return-Bytes, Query-Type, Tuple_Size).

Thus, as indicated in block **300-6**, the ODBC server component **20-12** writes statistics and user information into the tables contained in its memory area and into the log files in accordance with the Log **1** and Log **2** formats.

As indicated in FIG. **3**, blocks **304-1** through **304-4** indicate how the usage monitor component **20-6** performs its functions relative to creating entries in usage database **22**. As indicated in block **304-1**, for each end user session (end user connection), the ODBC server component **20-12** writes an entry into the SQL log **23** in the format described above. Subsequently, for each user that is logged on, for each SQL request, a log entry will be made to the SQL log **23** as indicated above. Since there are many users, these log entries will represent the random requests of many users. It is seen that the request information is only temporarily stored in the request table **200-3** contained in the ODBC server component's memory area. Once written to the log **23**, that entry in the SQL Request table **200-3** in the ODBC memory area is removed. The usage monitor component reads the ODBC SQL log **23** and writes data entries to a database record in a defined format as indicated in block **304-2**.

The usage monitor component **20-6** then runs reports to analyze the usage database as indicated in block **304-4**. Queries to summarize the entries by user, by time of day, by data warehouse table accessed, by query elapsed time, by row size returned, and by query type, are examples of reports that could be run.

Blocks **302-1** through **302-4** indicate how the SNMP agent component **20-2** carries out its function of providing updated user information. As indicated in block **302-1**, the SNMP agent component **20-2** sets an internal timer function to periodically poll the ODBC server component **20-12** via the API library component **20-8**. At each polling, the SNMP agent component **20-2** issues, for example, API call Get_Active_Connections that accesses that API library routine. This routine accesses the appropriate ODBC server memory table (ie., connections table **200-2**).

The ODBC server component **20-12** transfers the requested usage information received from its memory area to the SNMP agent component **20-2**. That is, the ODBC server component **20-12** obtains and returns the requested information to the SNMP agent component **20-2**.

Next, as indicated in block **302-2**, the SNMP agent component **20-2** transfers the usage information to the SNMP server component's MIB database **23** utilizing the SNMP protocoL More specifically, the SNMP agent component **20-2** issues a set command causing the information item(s) to be stored in the preallocated object areas of the MNB database **23** designated by the set command SNMP server component **30** in response to the set command performs the required operations for storing the usage information items. By way of example, these items could include objects defining the active connections, active servers and SQL requests issued by client end users that are formatted as indicated in the Appendix. As indicated in block **302-4**, the administration interface component **32** can be used to gather and report data from the MIB database **28**.

From the above, it is seen how the API of the present invention is able to provide interoperability between the monitoring and administration components of a data warehouse system utilizing an ODBC interface.

It will be appreciated that the teachings of the present invention may be used in conjunction with other types of data warehouse systems. Further, the present invention may be used with other types of application tools and interoperability protocols such as Java database connectivity protocols. Still further, the present invention may be incorporated into other types of data warehouse systems architectures. Many other changes will immediately occur to those skilled in the art.

APPENDICES

I. Glossary
II. MEB Objects and API

APPENDIX I

GLOSSARY

In the field of the present invention, the following terms have the following meanings:

1. API A set of routines used by an application program to direct the performance of procedures by the computer's operating system.
2. database management system (DBMS) A layer of software between the physical database and the user. The DBMS manages all requests for database action (for example, queries or updates) from the user. This eliminates the need for the user to keep track of the physical details of file locations and formats, indexing schemes, etc.
3. SQL Originally an acronym for Structured Query Language. Now the name of the language most commonly used to access relational databases.
4. administrator an individual who carries out tasks such as creating databases and/or monitoring the use and performance of those databases.
5. database A collection of data that has meaning to an organization or to an individual and that is managed as a unit.
6. SNMP The network management protocol of TCP/IP. In SNMP, agents monitor the activity in the various devices on the network and report to the network console workstation. Control information is maintained in a structure known as a management information base (MIB).
7. MIB A management information base comprises a set of objects describing software administrated by an SNMP manager such as the Bull ISM system or HP Openview system.
8. ODBC Open DataBase Connectivity specification provided by Microsoft Corporation that specifies an application interface to heterogeneous databases. The specification is implemented by various DBMS specific drivers that map the ODBC specification to the DBMS interface.
9. User A physical person or a unit in an enterprise. A user has a distinguished name, and is associated with "Authentication"attributes (e.g. password) and "privilege" attributes (e.g. role, category, classification, etc.).

APPENDIX II

```
A   Data Warehouse MIB Objects
    This section of the Appendix contains example definitions of the objects defined in
    a section of the Data Warehouse MIB 28 supported by the Data Warehouse
    SNMP agent component 20-2.
Active Connections
    NbActiveConnection OBJECT-TYPE
                SYNTAX Counter
                ACCESS read-only
                STATUS mandatory
           DESCRIPTION "Number of ODBC Server active connections"
                       ::= { ODBC 1 }
    ActiveCtionTable OBJECT-TYPE
                SYNTAX SEQUENCE OF ActiveConnectionEntry
                ACCESS not-accessible
                STATUS mandatory
           DESCRIPTION "Table containing information for each active
                       ODBC Server connection"
                       ::= { ODBC 2 }
    ActiveConnectionEntry OBJECT-TYPE
```

APPENDIX II-continued

```
                SYNTAX ActiveConnectionEntry
                ACCESS not-accessible
                STATUS mandatory
           DESCRIPTION "Information on one ODBC Server active
                       connection"
                 INDEX { ActiveCtionID },
                       ::= { ActiveCtionTable 1 }
ActiveConnectionEntry :: = SEQUENCE {
                       ActiveCtionID
                       Counter
                       ActiveCtionType
                       INTEGER
                       ActiveCtionLoginTime
                       DisplayString
                       ActiveCtionNBMsgSent
                       Counter
                       ActiveCtionNBMsgRcvd
                       Counter
                       ActiveCtionBytesRcvd
                       Counter
                       ActiveCtionLastMsgType
                       DisplayString
                       ActiveCtionLastMsgDir
                       INTEGER
                       }
ActiveCtionID OBJECT-TYPE
                SYNTAX Counter
                ACCESS read-only
                STATUS mandatory
           DESCRIPTION "Unique Identifier for one ODBC Server
                       active connection"
                       ::= { ActiveCtionEntry 1 }
ActiveCtionType OBJECT-TYPE
                SYNTAX INTEGER{
                       server (1),
                       client (2)
                       }
                ACCESS read-only
                STATUS mandatory
           DESCRIPTION "Connection type"
                       ::= { ActiveCtionEntry 2 }
ActiveCtionLoginTime OBJECT-TYPE
                SYNTAX DisplayString
                ACCESS read-only
                STATUS mandatory
           DESCRIPTION "The log-in time for this connection"
                       ::= { ActiveCtionEntry 3 }
ActiveCtionNBMsgSent OBJECT-TYPE
                SYNTAX Counter
                ACCESS read-only
                STATUS mandatory
           DESCRIPTION "Number of messages sent since the log-in
                       time of this connection"
                       ::= { ActiveCtionEntry 4 }
ActiveCtionNBMsgRcvd OBJECT-TYPE
                SYNTAX Counter
                ACCESS read-only
                STATUS mandatory
           DESCRIPTION "Number of messages received since the
                       log-in time of this connection"
                       ::= { ActiveCtionEtnry 5 }
ActionCtionBytesRcvd OBJECT-TYPE
                SYNTAX Counter
                ACCESS read-only
                STATUS mandatory
           DESCRIPTION "Number of bytes received since the log-in
                       time of this connection"
                       ::= { ActiveCtionEntry 6 }
ActiveCtionLastMsgType OBJECT-TYPE
                SYNTAX DisplayString
                ACCESS read-only
                STATUS mandatory
           DESCRIPTION "Type of the last message sent for this
                       ODBC Server active connection"
                       ::= { ActiveCtionEntry 7 }
ActiveCtionLastMsgDir OBJECT-TYPE
                SYNTAX INTEGER{
                       from (1),
                       to (2)
```

APPENDIX II-continued

```
                }
        ACCESS read-only
        STATUS mandatory
   DESCRIPTION "Direction of the last message sent for
                this active connection (from or to this
                ODBC Server)"
                ::= { ActiveCtionEntry 8 }
```

The MIB section also includes object definitions for the various items included in the servers table and SQL requests table of FIG. 2.

```
B. Application Programming Interface (API) for Monitoring Data Warehouse
   Activity
   This section of the Appendix describes the API used by the SNMP Agent to access
   the ODBC Server Administration information.
   1) GET_SERVER_PARAMETERS(
                PROCESS_ID,
                LISTEN_ADDRESS,
                SERVER_NAME,
                RETURN_CODE)
   2) GET_ACTIVE_CONNECTIONS(
                SESSION_ID,
                CONNECT_ID,
                USER_ID,
                DATABASE_NAME,
                LOGIN_DATE_TIME,
                LOGOUT_DATE_TIME,
                NUM_MESSAGES_SENT,
                NUM_MESSAGES_RECEIVED,
                NUM_BYTES_RECEIVED,
                LAST_MESSAGE,
                DIRECTION_OF_LAST_MESSAGE,
                RETURN_CODE)
   3) GET_ACTIVE_SQL_REQUESTS(
                SESSION_ID,
                SQL_TEXT,
                QUERY_START_TIME,
                QUERY_STOP_TIME,
                RETURN_STATUS,
                NUM_ROWS_RETURNED,
                NUM_BYTES_RETURNED,
                QUERY_TYPE,
                TUPLE_SIZE,
                RETURN_CODE)
```

One row is returned with each function call the function should be called repeatedly until return code EOF (end of file) is encountered.

While in accordance with the provisions and statutes there has been illustrated and described the best form of the invention, certain changes may be made without departing from the spirit of the invention as set forth in the appended claims and that in some cases, certain features of the invention may be used to advantage without a corresponding use of other features.

What is claimed is:

1. A method for facilitating interoperability between components of a data warehouse system containing a warehouse database for storing warehouse information, the components including a number of different monitoring and administration components for monitoring users and recording information relating to the activity of warehouse processes pertaining to accessing information stored in the warehouse database, the method comprising:

(a) including in the warehouse system, an ODBC server component operatively coupled to a number of ODBC client systems for receiving SQL requests through a first standard protocol used for data connectivity, the ODBC server component being operatively coupled to the warehouse database through an ODBC driver component for accessing information from the warehouse database using the first standard data connectivity protocol;

(b) including in the warehouse system, a storage log facility operatively coupled to the ODBC server component and to a predetermined one of the different warehouse components for enabling storage of information pertaining to user sessions and SQL queries by the OBDC server component for optimizing warehouse database storage and interfaces; and,

(c) including in the warehouse system an API component as part of the ODBC server component that provides interoperability between the first standard protocol and other standard protocols for enabling the different warehouse monitoring and administration components to perform their functions pertaining to the warehouse database utilizing information received from the ODBC client systems and stored in the storage log facility.

2. The method of claim 1 wherein the method further comprises the step of including in the number of different monitoring and administration warehouse components:

(d) a network agent component for performing the function of monitoring users and the activity of warehouse processes utilizing one of the other standard protocols; and,

(e) an usage monitoring component coupled to the network agent component, the usage monitoring component for performing the function of recording in a usage database, information pertaining to user sessions and SQL queries issued by the individual users of the client systems to access the warehouse database.

3. The method of claim 2 further including the step of populating the usage database with records having a predefined format by the usage monitoring component accessing information from the storage log facility.

4. The method of claim 1 wherein step (b) further includes:

storing the information as entries having a predetermined format into a number of different log files.

5. The method of claim 4 wherein the method further includes the steps of:

storing entries in a first file of the number of different log files corresponding to records identifying user sessions and their attributes; and

storing entries in a second file of the number of different log files corresponding to records identifying user SQL statements and their attributes.

6. The method of claim 5 wherein each record stored in the first log file is formatted to include the following information: User_name; Database_name; Login_date; Logout-date; Session_ID; and wherein each record of the second log file is formatted to include the following information:

User Requests; Session_ID; SQL-text; Query-time; Return_status; Return_rows; Return_bytes; Query-type and Tuple-size.

7. The method of claim 1 wherein the method further comprises the step of including in the ODBC server component, an allocated memory area for storing routines included in the API component for maintaining interoperability between warehouse components and a number of table structures for storing entries pertaining to tracking servers operation, active end user database connections and end user SQL requests.

8. The method of claim 7 wherein the routines of the API component includes a first routine for obtaining parameters for the ODBC server component and warehouse server components, a second routine for obtaining information pertaining to active end user connections and a third routine for obtaining information pertaining to active SQL query requests made by end users.

9. The method of claim 7 wherein the number of table structures includes a servers table, a connections table and an SQL requests table.

10. The method of claim 7 wherein the servers table includes the following information sections: a process ID section, a listen address section and a server name section.

11. The method of claim 7 wherein the connections table includes the following information sections: session ID, connect ID, user ID, database name, login date/time; logout date/time, number of messages sent, number of messages received, number of bytes received, last message and the direction of the last message.

12. The method of claim 7 wherein the SQL requests table includes the following information sections: session ID, SQL text, query start time, query stop time, return status, number of rows returned, number of bytes returned, query type and tuple size.

13. The method of claim 2 wherein the usage monitoring component performs the functions of reading the storage log facility and writing data records in a defined format and running reports for analyzing the usage database.

14. The method of claim 7 wherein the number of different monitoring and administration warehouse components further includes an SNMP server component that operatively couples to the network agent component corresponding to an SNMP agent component and includes a MIB database for storing a number of information objects, the method further including the step of periodically polling the ODBC server component by the SNMW server component through the API component routines utilizing the one of the standard protocol corresponding to an SNMP protocol for obtaining current usage information from the table structures of the allocated memory area of the ODBC server component for transfer to a section of the MIB database allocated for monitoring data warehouse activity.

15. The method of claim 14 wherein the section is organized to contain objects being managed by the SNMP server component defining active connections, active servers and SQL requests issued by client end users.

16. The method of claim 15 wherein the method further comprises the step of including an administration interface in the SNMP server component for enabling an administrator to gather and report warehouse data activity derived from the objects stored in the MIB database.

17. The method of claim 16 wherein the method further comprises the step of including enabling the starting and stopping of the ODBC server component and for operating different ones of the warehouse components of the warehouse system.

18. A facility for providing interoperability between components of a data warehouse system containing a warehouse database for storing warehouse information, the components including a number of different monitoring and administration components for monitoring users and recording information relating to the activity of warehouse processes pertaining to accessing information stored in the warehouse database, the facility comprising:

(a) an ODBC server component operatively coupled to a number of ODBC client systems for receiving SQL requests through a first standard protocol used for data connectivity, the ODBC server component being operatively coupled to the warehouse database through an ODBC driver component for accessing information from the warehouse database using the first standard data connectivity protocol;

(b) a storage log facility operatively coupled to the ODBC server component and to a predetermined one of the warehouse components for enabling storage of information pertaining to user sessions and SQL queries by the ODBC server component for optimizing warehouse database storage and interfaces; and,

(c) an API component included as part of the ODBC server component that provides interoperability between the first standard protocol and other standard protocols for enabling the different warehouse monitoring and administration components to perform their functions relating to the warehouse database utilizing information received from the ODBC client systems and stored in the storage log facility.

19. The facility of claim 18 wherein the number of different monitoring and administration warehouse components includes:

(a) a network agent component for performing the function of monitoring users and the activity of warehouse processes utilizing one of the other standard protocols; and,

(b) an usage monitoring component coupled to the network agent component, the usage monitoring component recording in a usage database, information pertaining to user sessions and SQL queries issued by the individual users of the client systems to access the warehouse database.

20. The facility of claim **19** wherein the functions performed by the usage monitor component operates to access information from the storage log facility to populate the usage database with records having a predefined format.

21. The facility of claim **18** wherein the ODBC server component stores the information as entries having a predetermined format into a number of different log files.

22. The facility of claim **21** wherein the number of log files includes:

a first file for storing entries corresponding to records identifying user sessions and their attributes; and

a second file for storing entries corresponding to records identifying user SQL statements and their attributes.

23. The facility of claim **22** wherein each record stored in the first log file is formatted to include the following information: User_name; Database_name; Login-date; Logout-date; Session_ID; and wherein each record of the second log file is formatted to include the following information:

User_Requests; Session-ID; SQL text; Query-time; Return_status; Return_rows; Return-bytes; Query-type and Tuple_size.

24. The facility of claim **18** wherein the ODBC server component further includes an allocated memory area for routines included in the API component for maintaining interoperability between warehouse components and a number of table structures for storing entries pertaining to tracking servers operation, active end user database connections and end user SQL requests.

25. The facility of claim **24** wherein the routines of the API component includes a first routine for obtaining parameters for the ODBC server component and warehouse server components, a second routine for obtaining information pertaining to active end user connections and a third routine for obtaining information pertaining to active SQL query requests made by end users.

26. The facility of claim **24** wherein the number of table structures includes a servers table, a connections table and an SQL requests table.

27. The facility of claim **26** wherein the servers table includes the following information sections: a process ID section, a listen address section and a server name section.

28. The facility of claim **26** wherein the connections table includes the following information sections: session ID, connect ID, user ID, database name, login date/time; logout date/time, number of messages sent, number of messages received, number of bytes received, last message and the direction of the last message.

29. The facility of claim **26** wherein the SQL requests table includes the following information sections: session ID, SQL text, query start time, query stop time, return status, number of rows returned, number of bytes returned, query type and tuple size.

30. The facility of claim **19** wherein the usage monitoring component performs the functions of reading the storage log facility and writing data records in a defined format and running reports for analyzing the usage database.

31. The facility of claim **24** wherein the number of different monitoring and administration warehouse components further includes an SNMP server component that operatively couples to the network agent component corresponding to an SNMP agent component and includes a MEB database for storing a number of information objects, the SNMP agent component being operative to periodically poll the ODBC server component through the API component routines utilizing the one of the standard protocols corresponding to an SNMP protocol for obtaining current usage information from the table structures of the allocated memory area of the ODBC server component for transfer to a section of the MEB database allocated for monitoring data warehouse activity.

32. The facility of claim **31** wherein the section of the NMB database is organized to contain objects being managed by the SNMP server component defining active connections, active servers and SQL requests issued by client end users.

33. The facility of claim **32** wherein the SNMP server component further includes an administration interface for enabling an administrator t warehouse data activity derived from the objects stored in the MEB database.

34. The facility of claim **33** wherein the administration interf facilities for enabling the starting and stopping of the ODBC server component and for operating different ones of the warehouse components of the warehouse system.

* * * * *